Introducing F#

Don Syme
MSR Cambridge

Introducing F#

Don Syme
MSR Cambridge

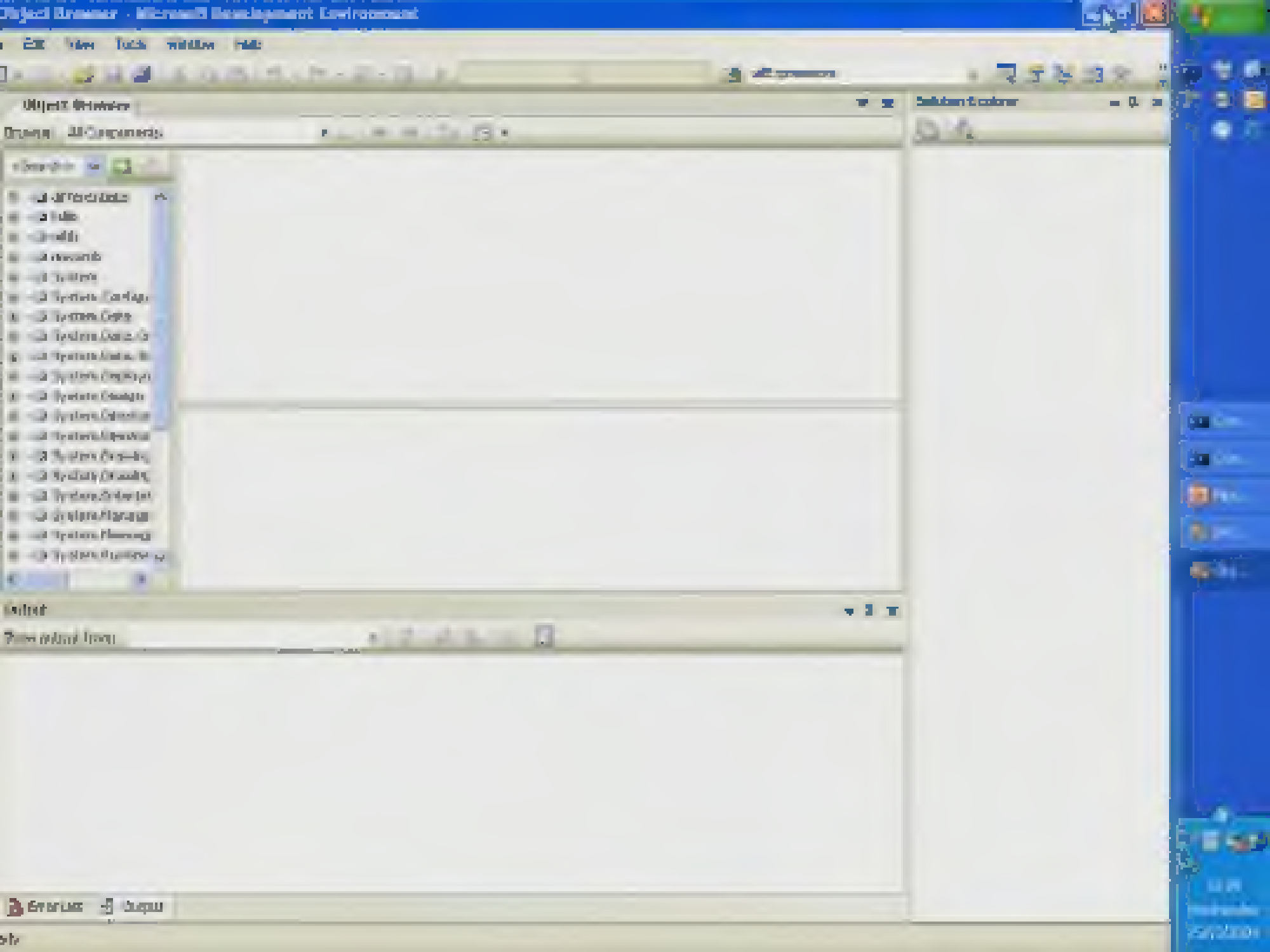
Outline

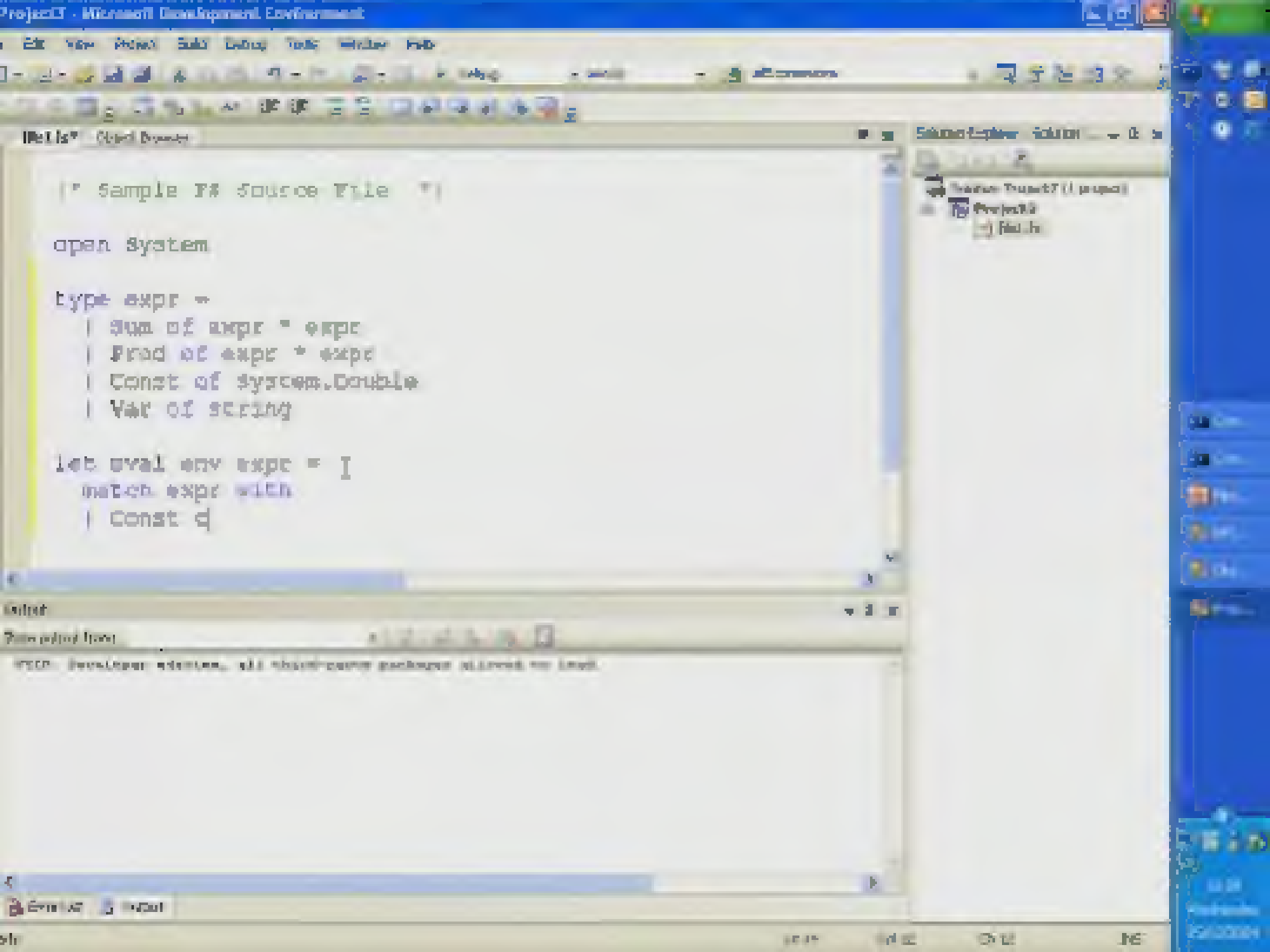
- F#: a quick orientation via a demo
- F#: orthogonal, simple features
- F#: co-operates in the context of .NET

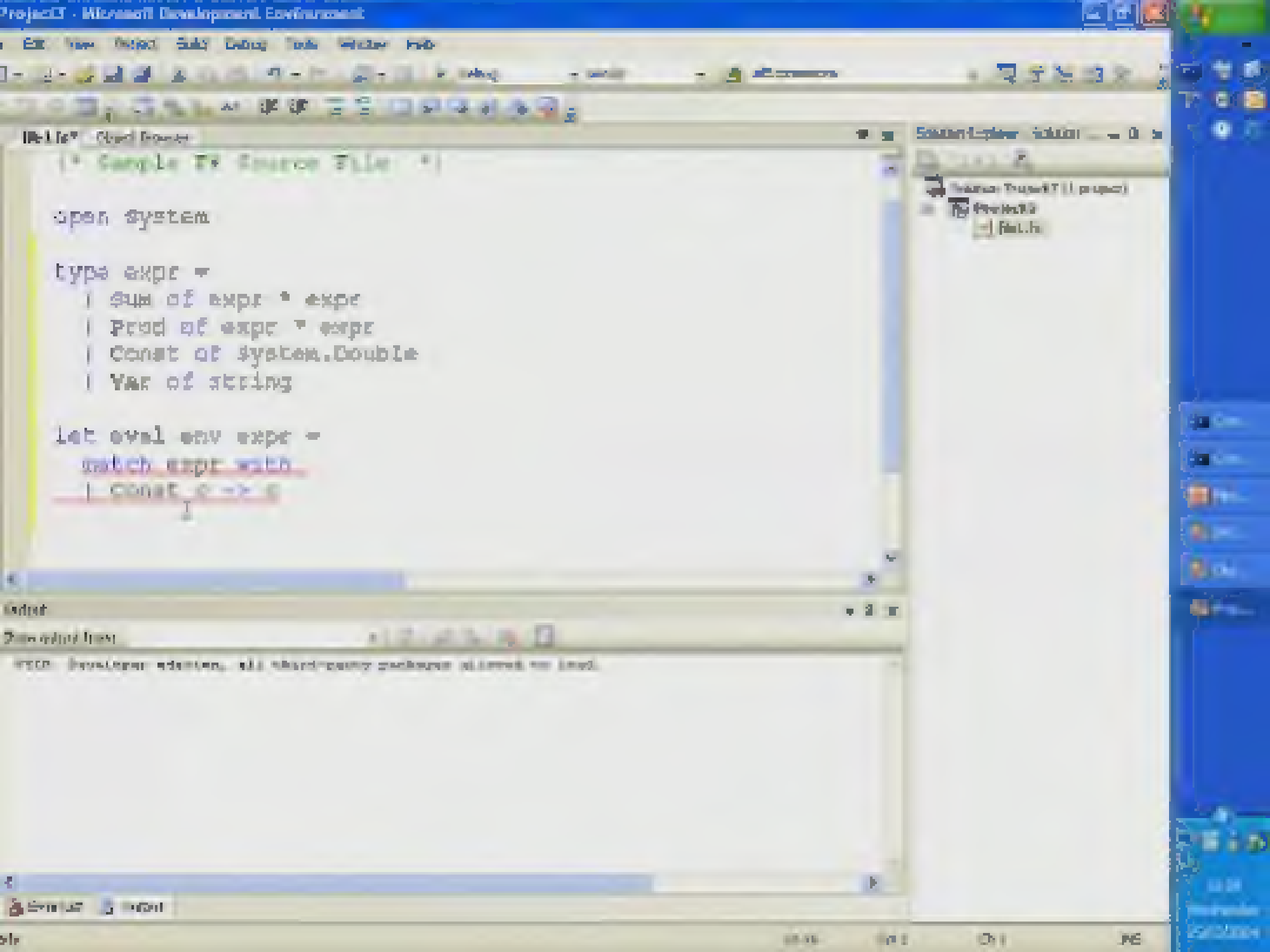
An ML-F# Orientation

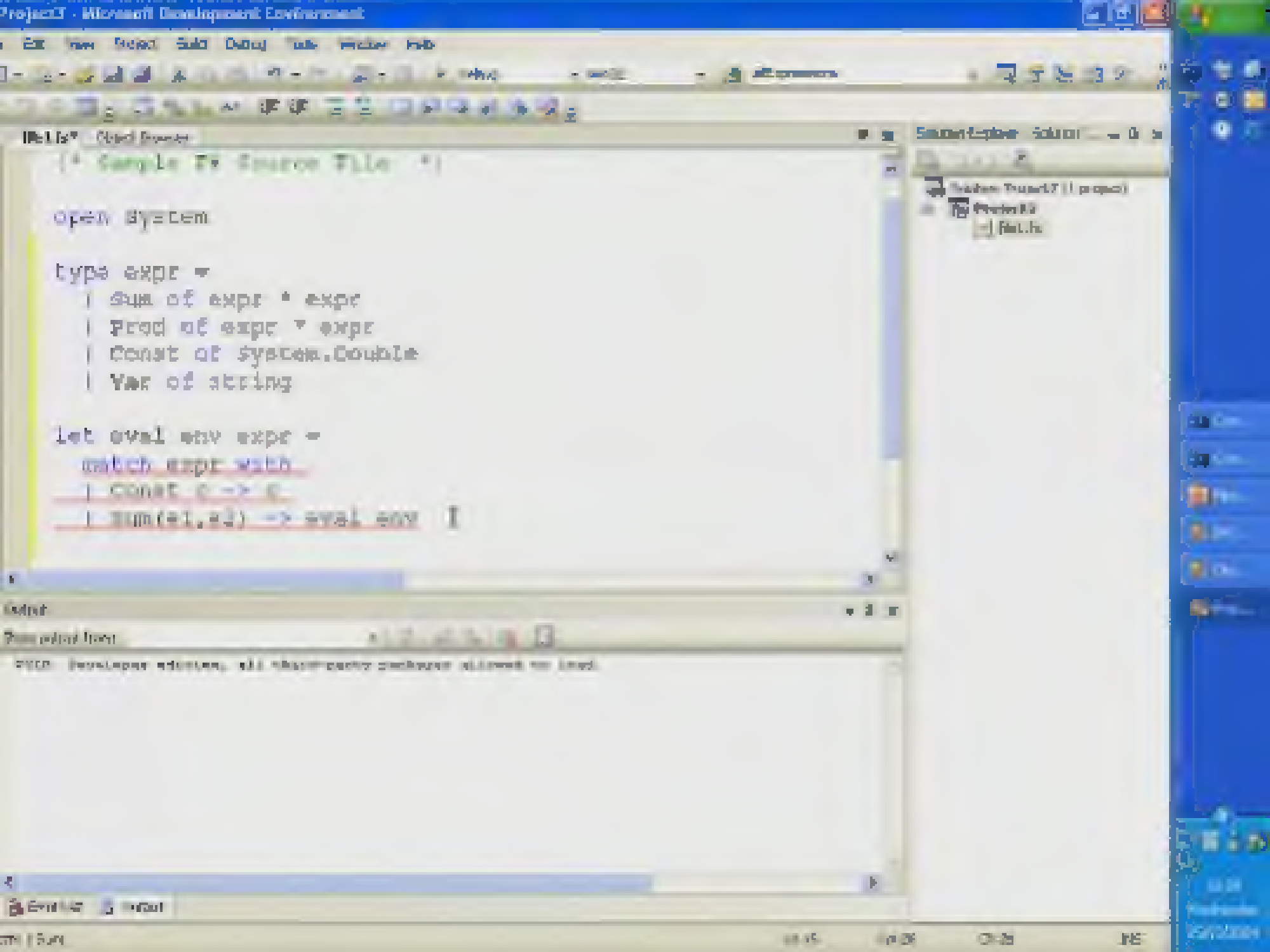
- DEMO

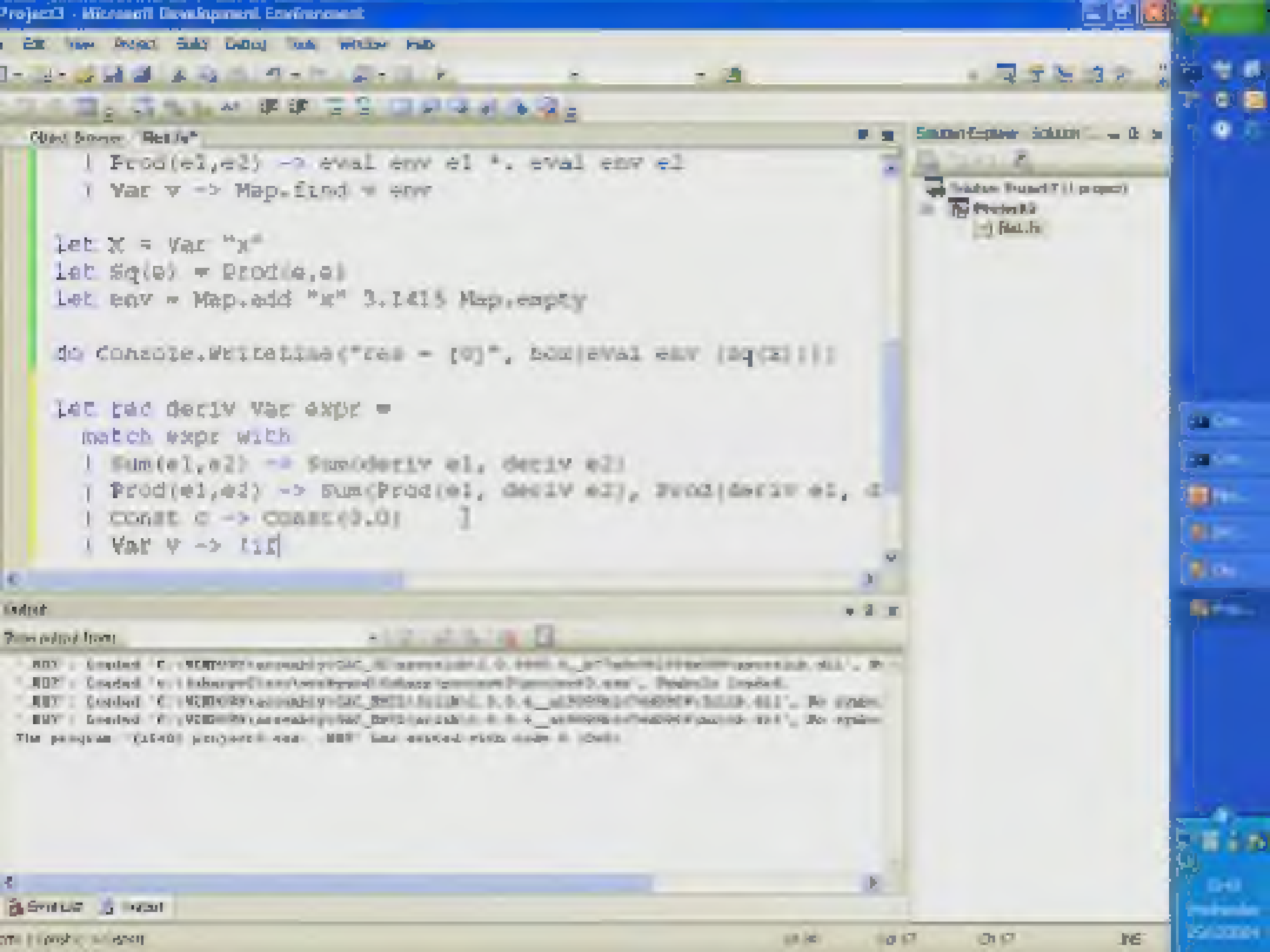
- 1. Write a symbolic differentiator
- 2. Write an evaluation function
- 3. Make it extensible with new operators
- 4. Display sample plots on a form
- 5. Use the differentiator from C#

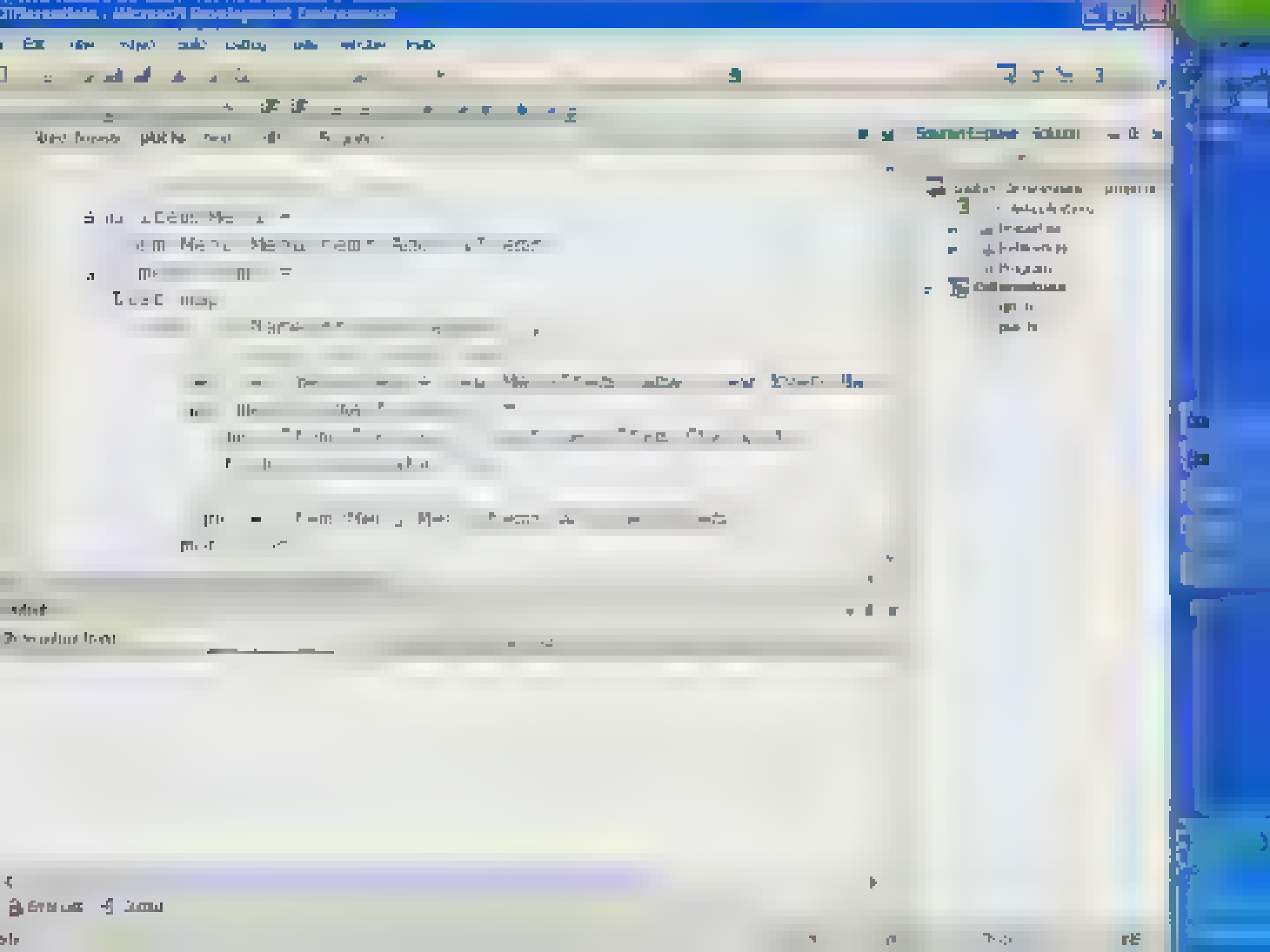


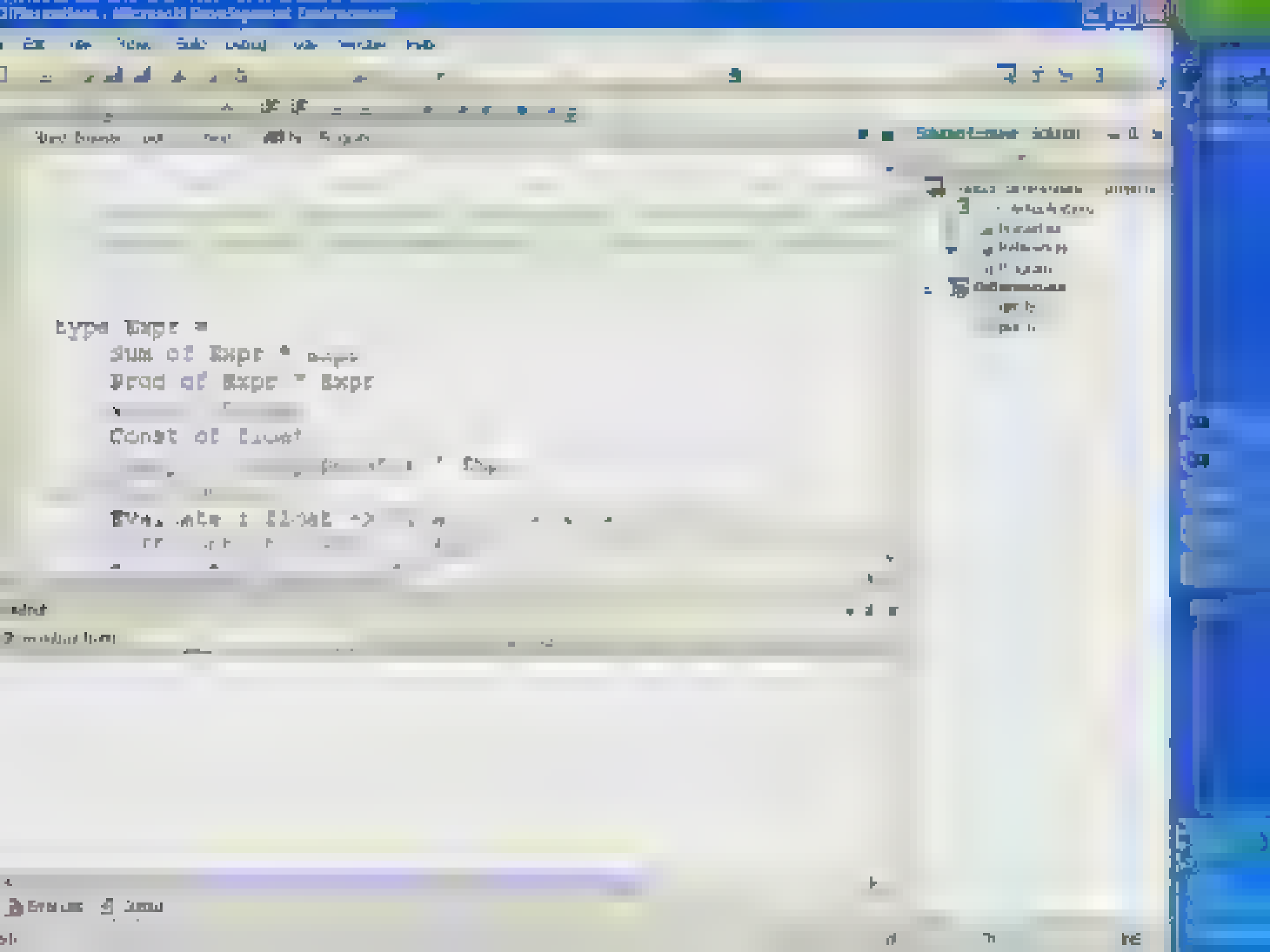

(* Sample F# Source File *)
open System
type expr =
| Sum of expr * expr
| Prod of expr * expr
| Const of System.Double
| Var of string
let eval env expr =
match expr with
| Const c

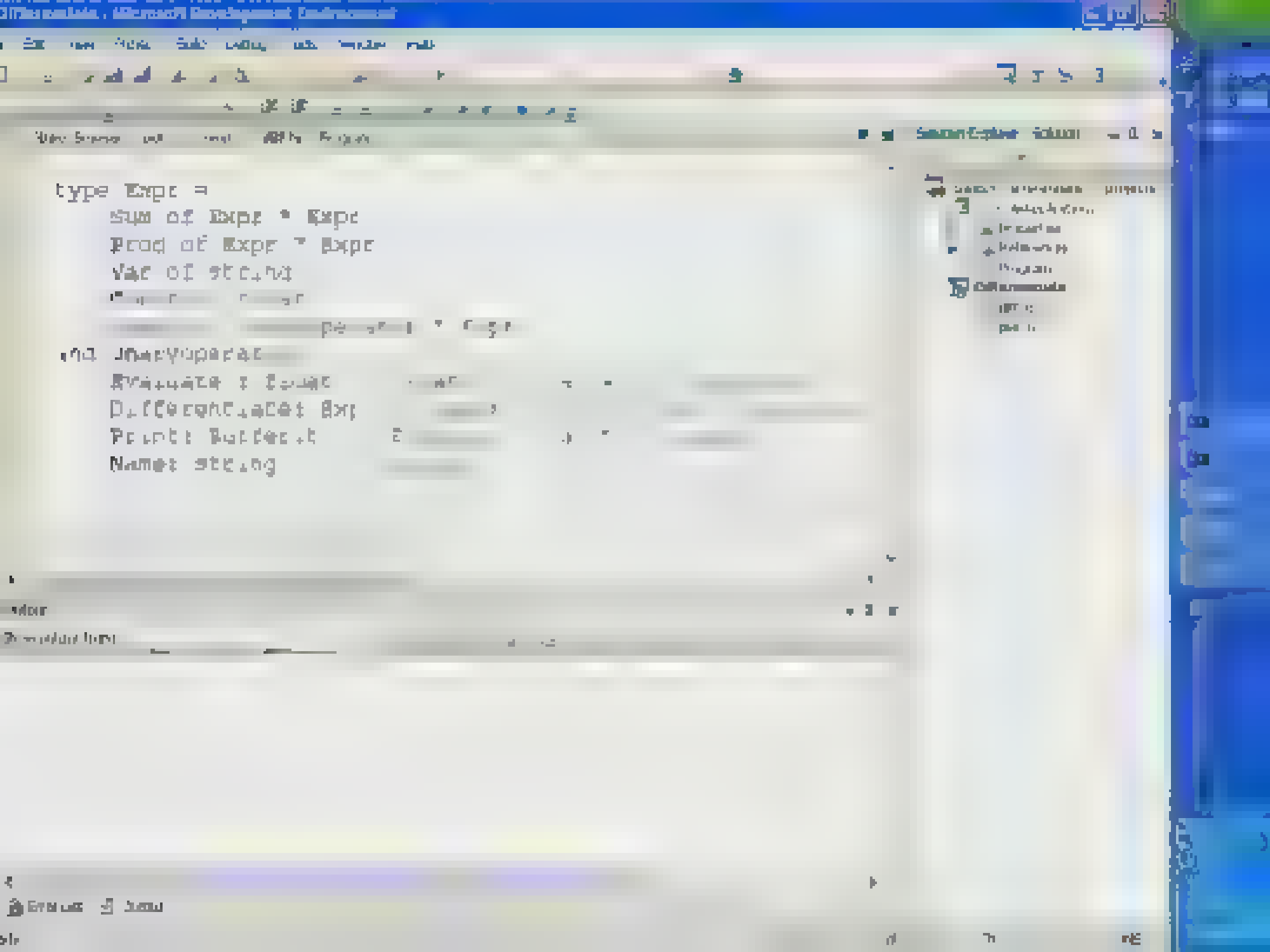

// Sample F# Source File
open System
type expr =
| Sum of expr * expr
| Prod of expr * expr
| Const of System.Double
| Var of string
let eval env expr =
match expr with
| Const c -> c

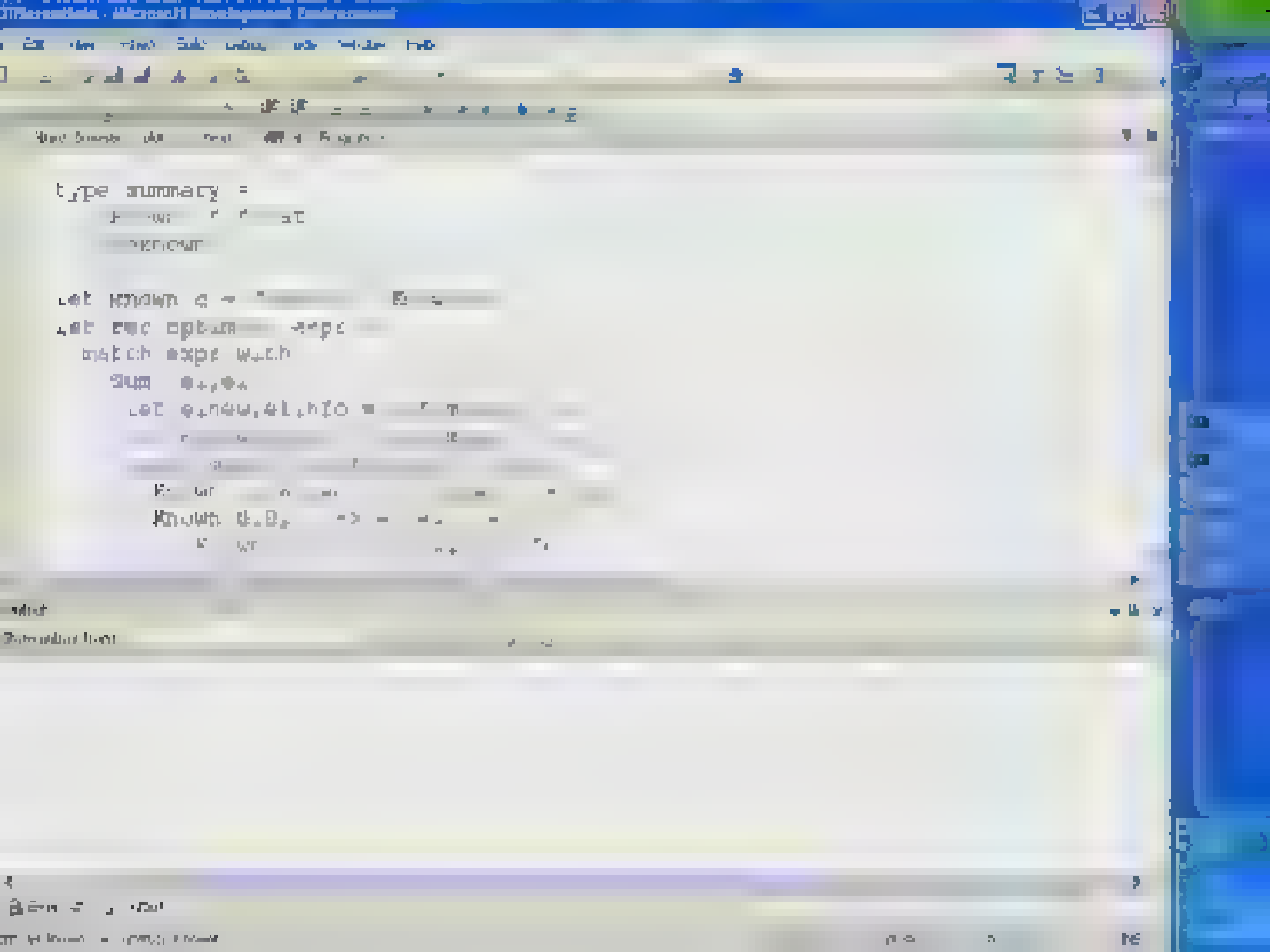

(* Sample F# Source File *)
open System
type expr =
| Sum of expr * expr
| Prod of expr * expr
| Const of System.Double
| Var of string
let eval env expr =
match expr with
| Const c -> c
| Sum(e1,e2) -> eval env

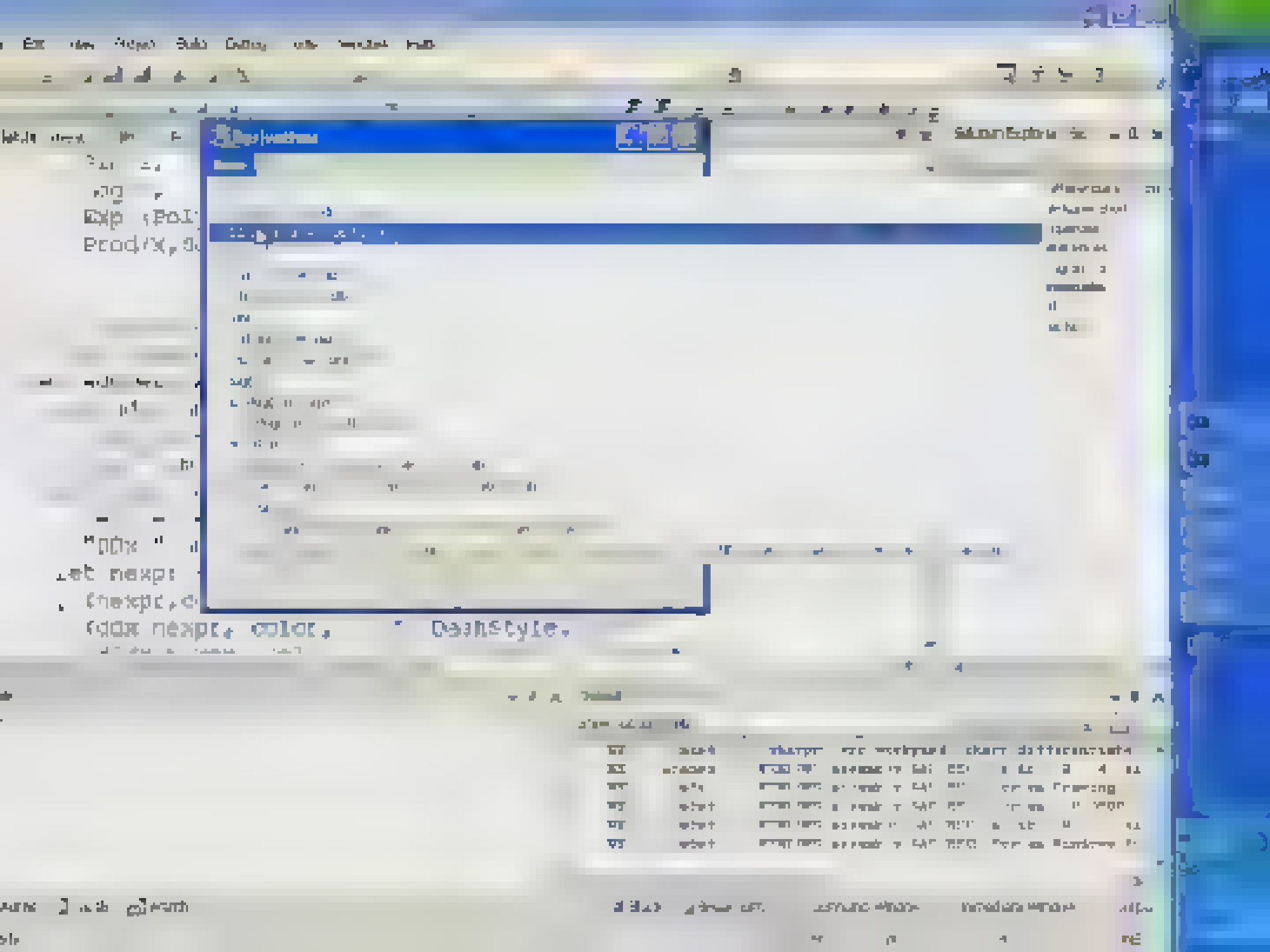

| Prod(e1,e2) -> eval env e1 *. eval env e1
| Var v -> Map.find v env
let x = Var "x"
let sq(e) = Prod(e,e)
let env = Map.add "x" 3.141 Map.empty
do Console.WriteLine("res = {0}", box(eval env (sq(x))))
let rec deriv var expr =
match expr with
| Sum(e1,e2) -> Sum(deriv e1, deriv e2)
| Prod(e1,e2) -> Sum(Prod(e1, deriv e2), Prod(deriv e1,
| Const c -> Const(0.0)
| Var v -> if

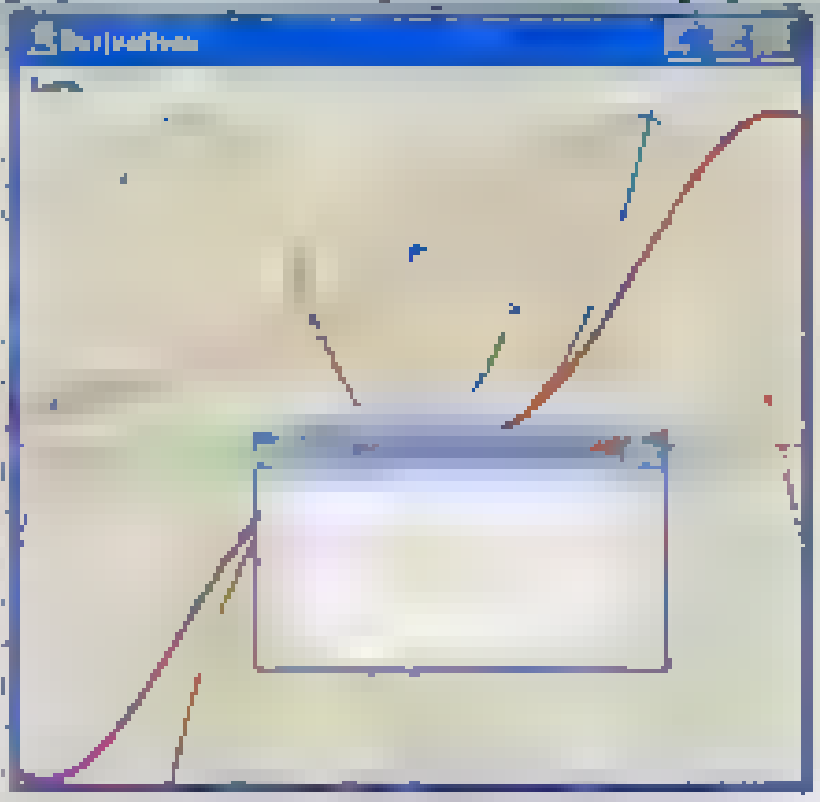


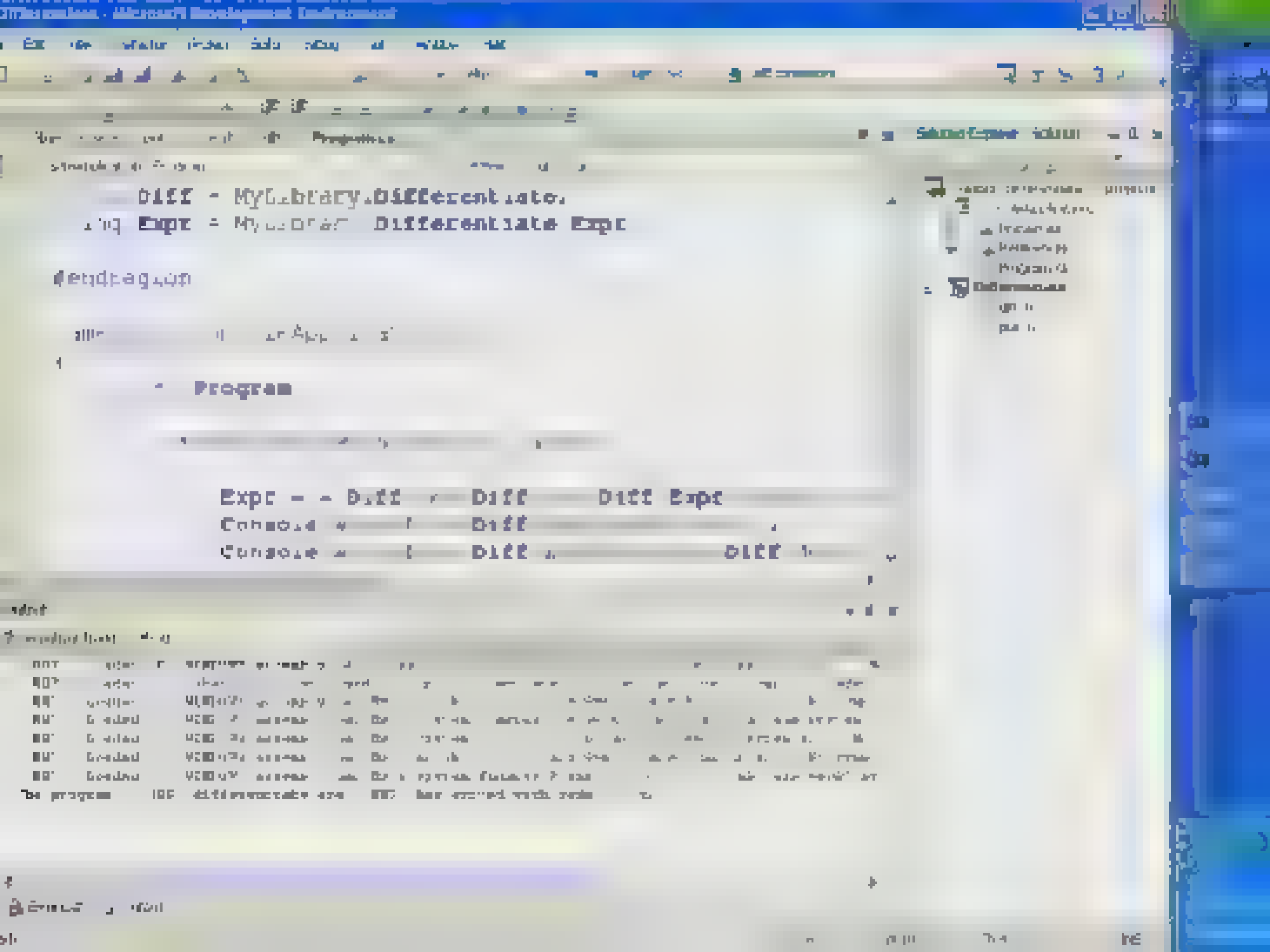

type Expr =
Sum of Expr *
Prod of Expr * Expr
Const of Int

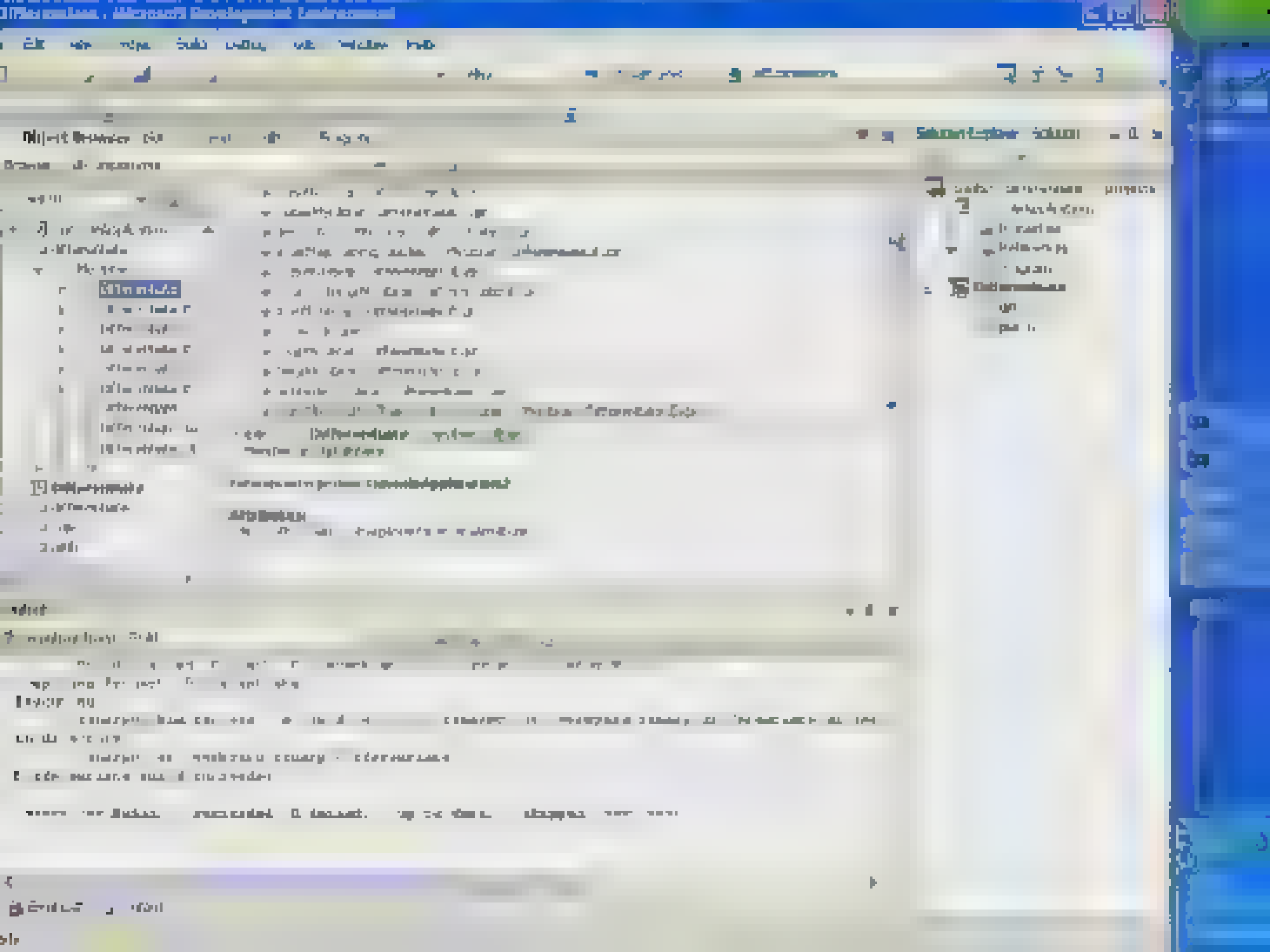

type Expr =
Sum of Expr * Expr
Prod of Expr * Expr
Var of string








Diff = MyLibrary.Differentiate;
Expr = MyLibrary.Differentiate.Expr
Program
Expr = - Diff Diff Diff Expr



The ML Family

- Static analysis of programs
- Program transformation
- Compilers
- Verification: model checking, theorem proving

ML in Microsoft

- Static Driver Verifier (Core OS)
- High-level model of FRS (Core File Services)
- Zap theorem prover (MSR)
- KIS race condition finder (MSR)

F# within .NET

Profilers
Optimizers etc

CLR GC.
JIT NGEN etc

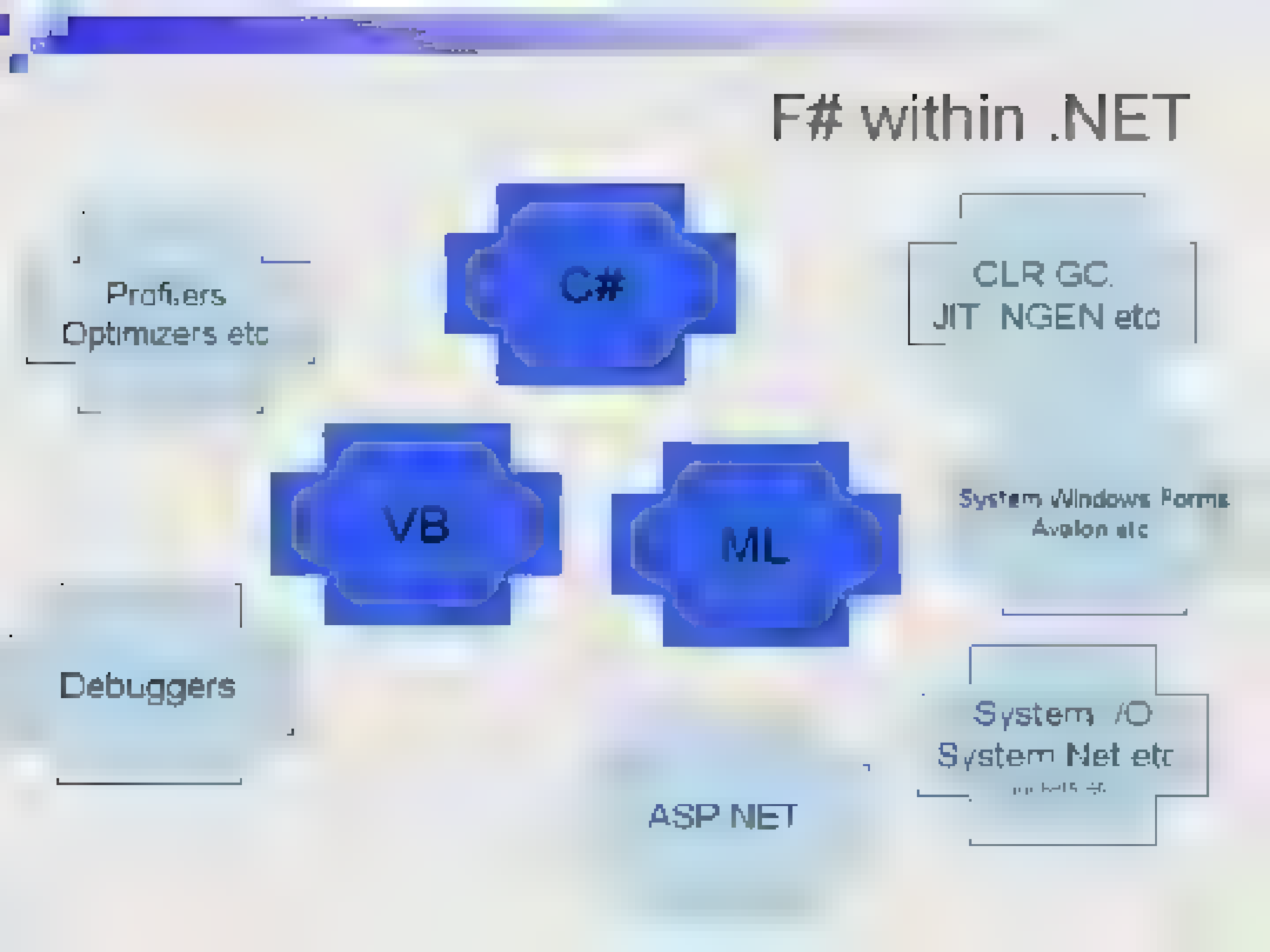
System.Windows.Forms
Avalon etc

Debuggers

System. IO
System.Net etc

ASP.NET

F# within .NET


Profilers
Optimizers etc
C#
CLR GC
JIT NGEN etc
VB
ML
System.Windows.Forms
Avalon etc
Debuggers
System.IO
System.Net etc
ASP.NET

F# as a Language

OCaml

F#

F# as a Language

OCaml

F#

F# as a Language

OCaml

F#

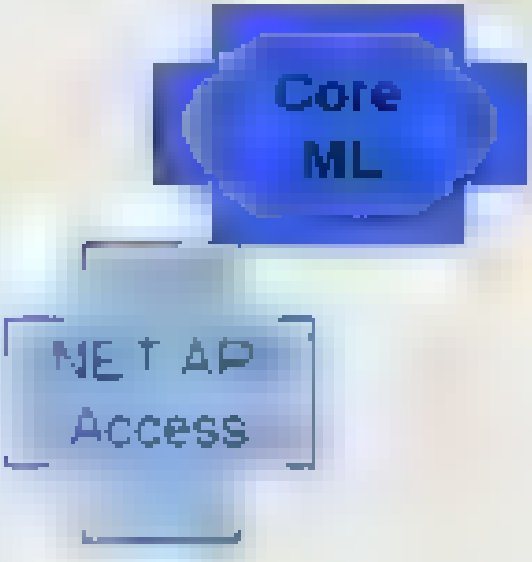
F# as a Language

Modules-as-values

OCaml

F#

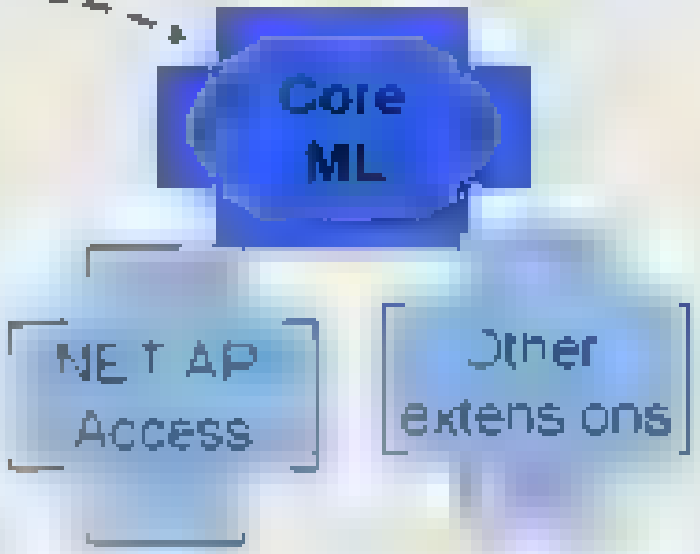
F# as a Language



OCaml + tools F#

F# as a Language

Powerful, simple programming language

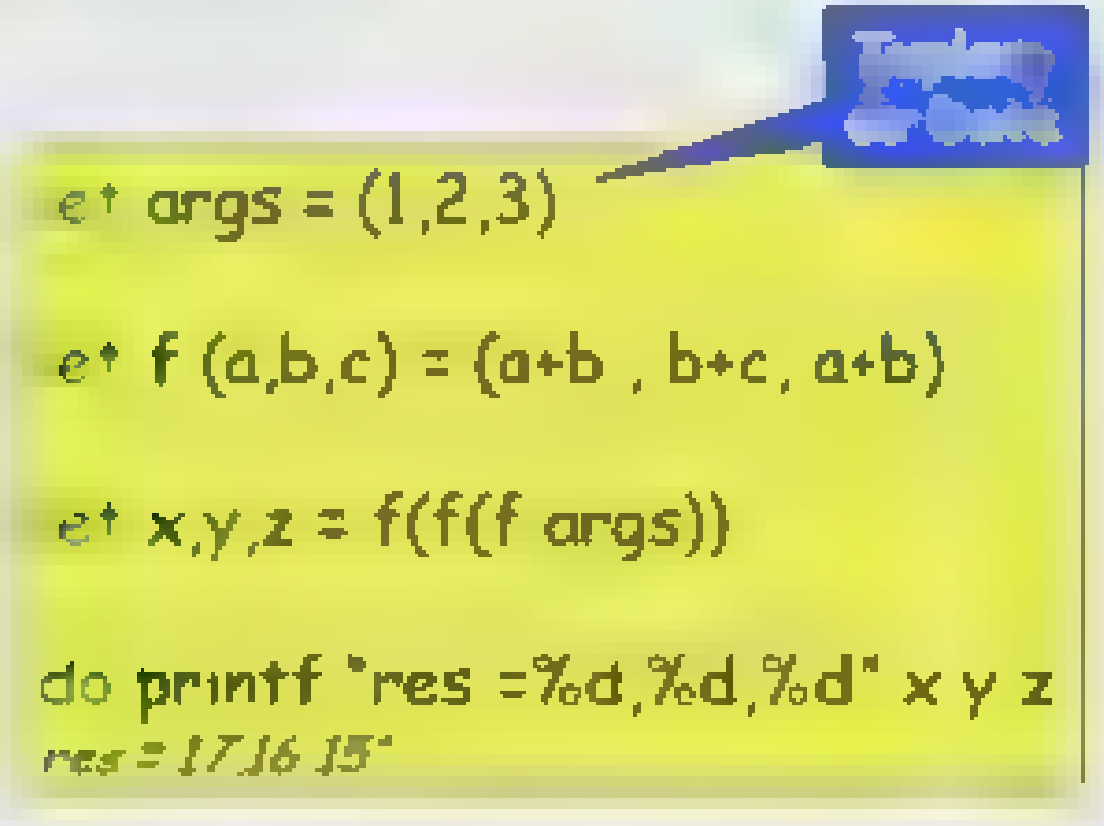


OCaml + tools

F# + tools

Orthogonal & Unified Constructs

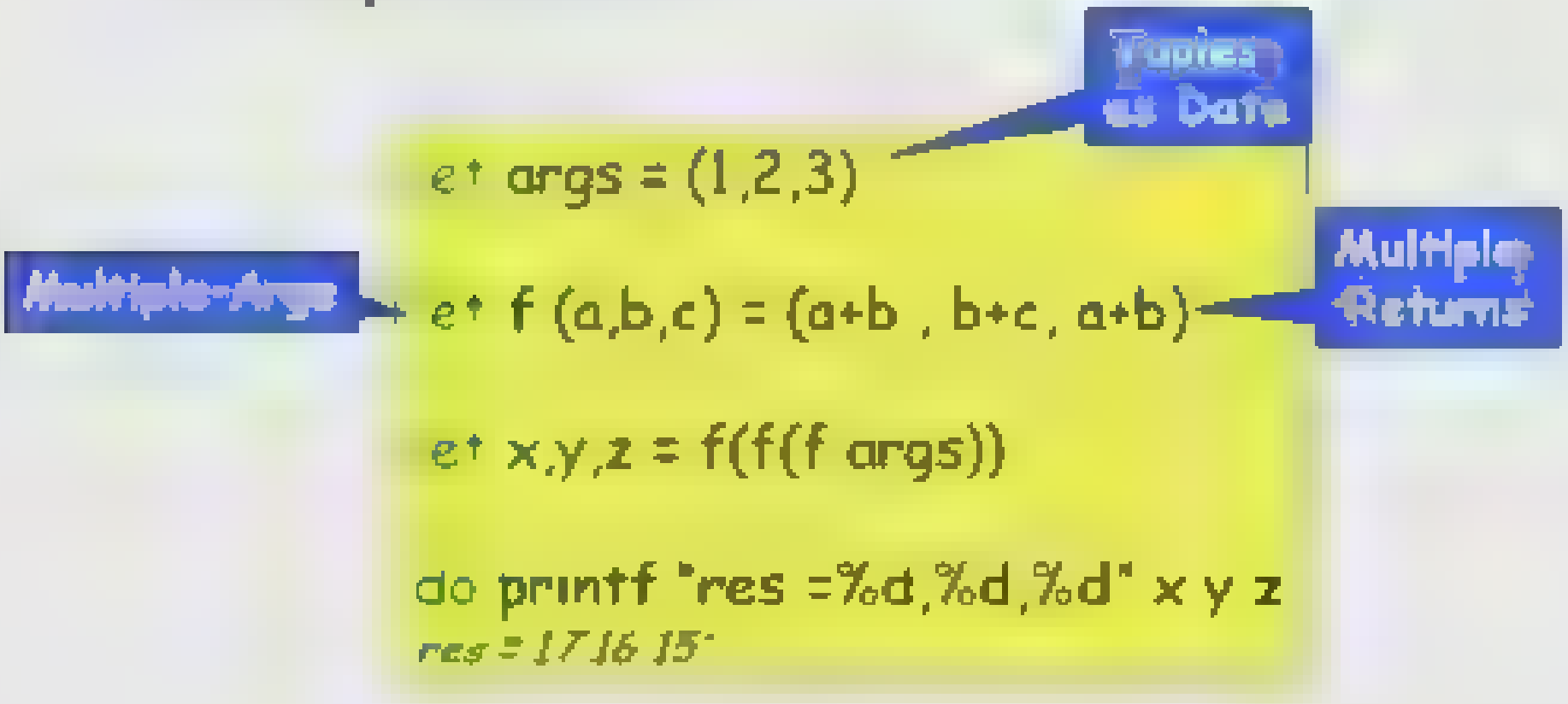
- Tuples = multiple arguments = multiple returns



```
e↑ args = (1,2,3)

e↑ f (a,b,c) = (a+b , b+c, a+b)

e↑ x,y,z = f(f(f args))

do printf "res =%d,%d,%d" x y z
res = 17 16 15"
```

Orthogonal & Unified Constructs

- Tuples = multiple arguments = multiple returns



Orthogonal & Unified Constructs

- Let "let" simplify your life...

```
let data = (1,2,3)

let f(a,b,c) =
    let sum = a + b + c in
    let g(x) = sum + x*x in
    g(a), g(b), g(c)
```

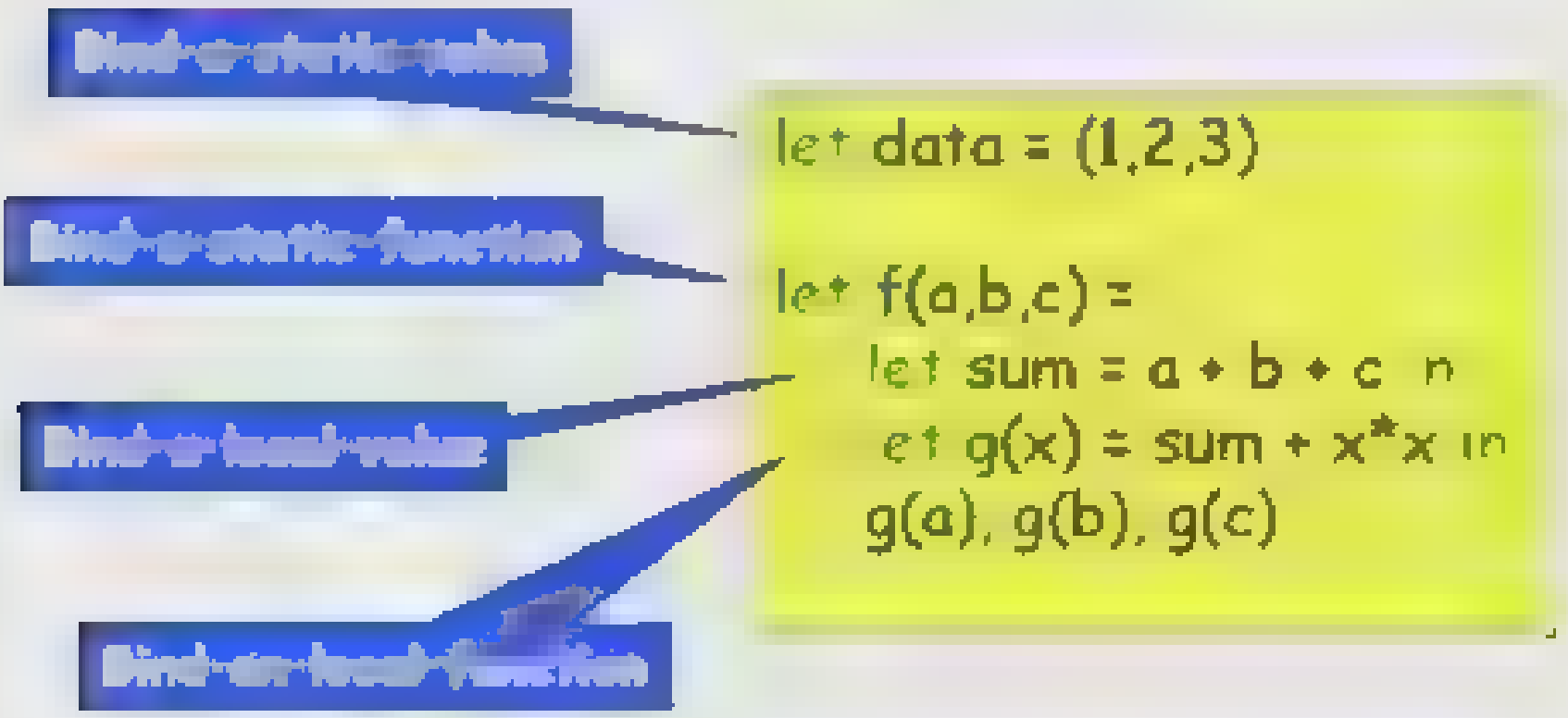
Orthogonal & Unified Constructs

- Let "let" simplify your life ...

```
let data = (1,2,3)

let f(a,b,c) =
    let sum = a + b + c in
    let g(x) = sum + x*x in
    g(a), g(b), g(c)
```

Orthogonal & Unified Constructs

- Let "let" simplify your life...



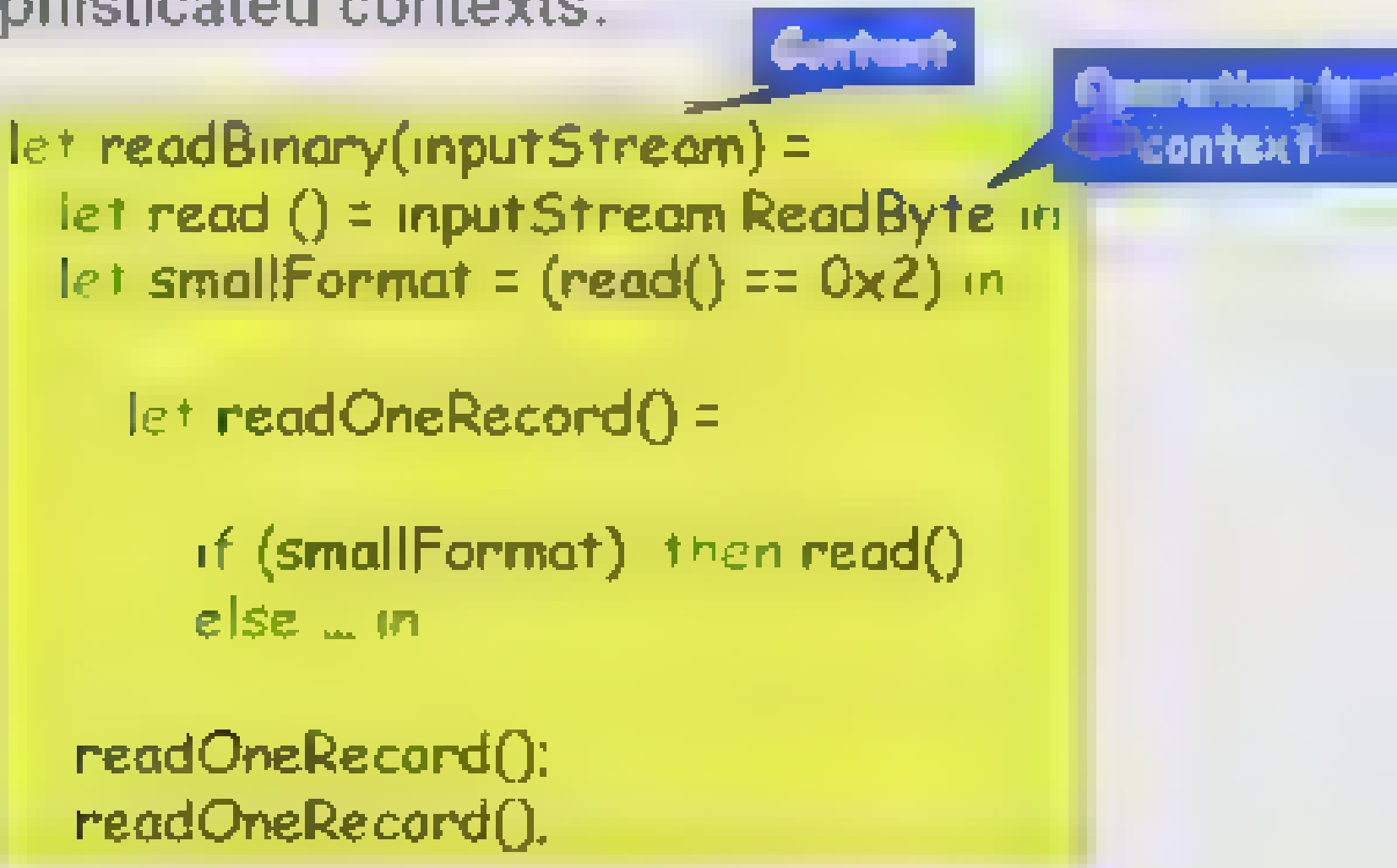
Orthogonal & Unified Constructs

- "let" + "capture": sophisticated operations in sophisticated contexts

```
let readBinary(inputStream) =
  let read () = inputStream ReadByte in
  let smallFormat = (read() == 0x2) in

    let readOneRecord() =

      if (smallFormat) then read()
      else ... in

  readOneRecord().
  readOneRecord().
```

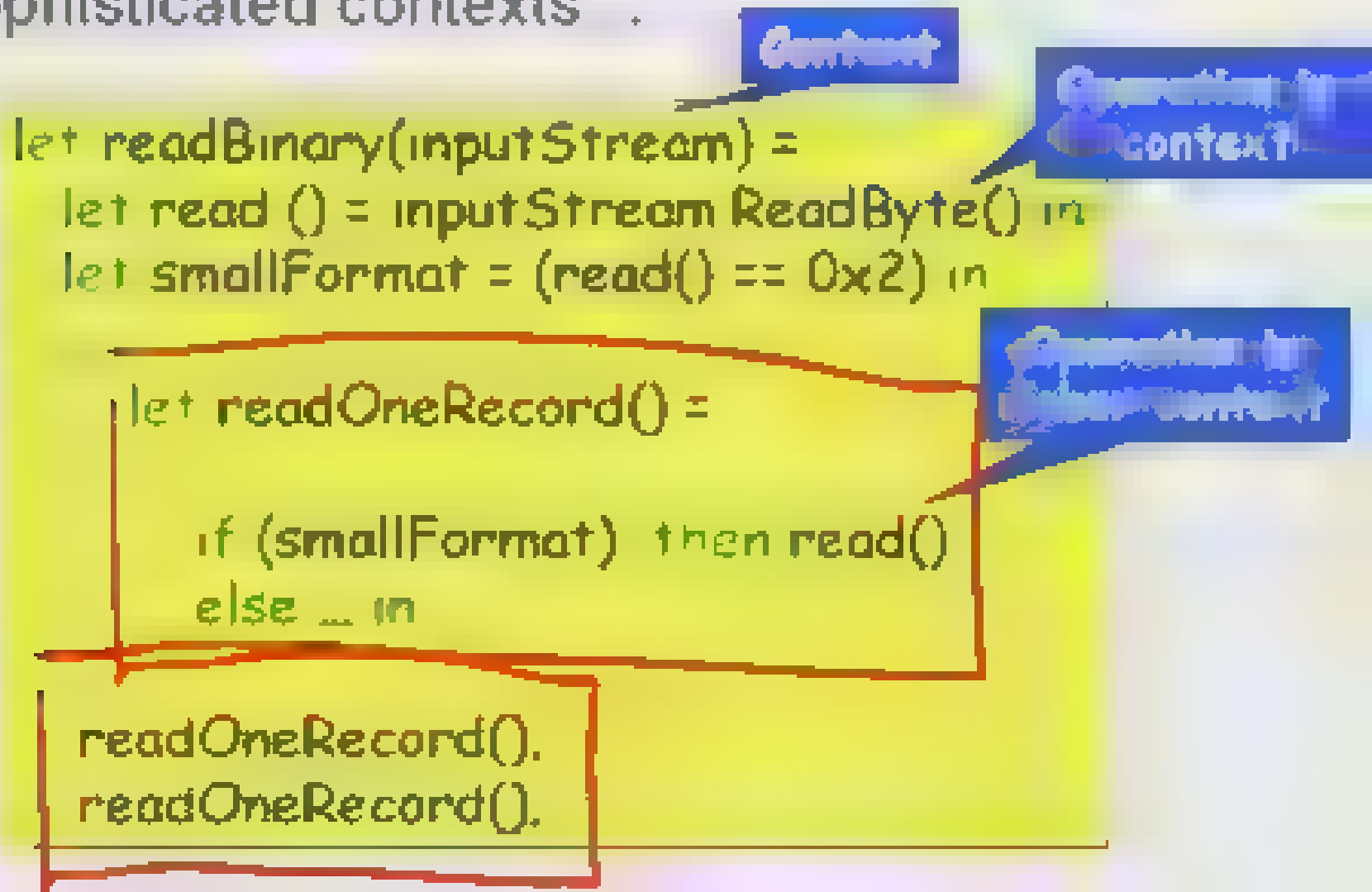
Orthogonal & Unified Constructs

- "let" + 'capture": sophisticated operations in sophisticated contexts.



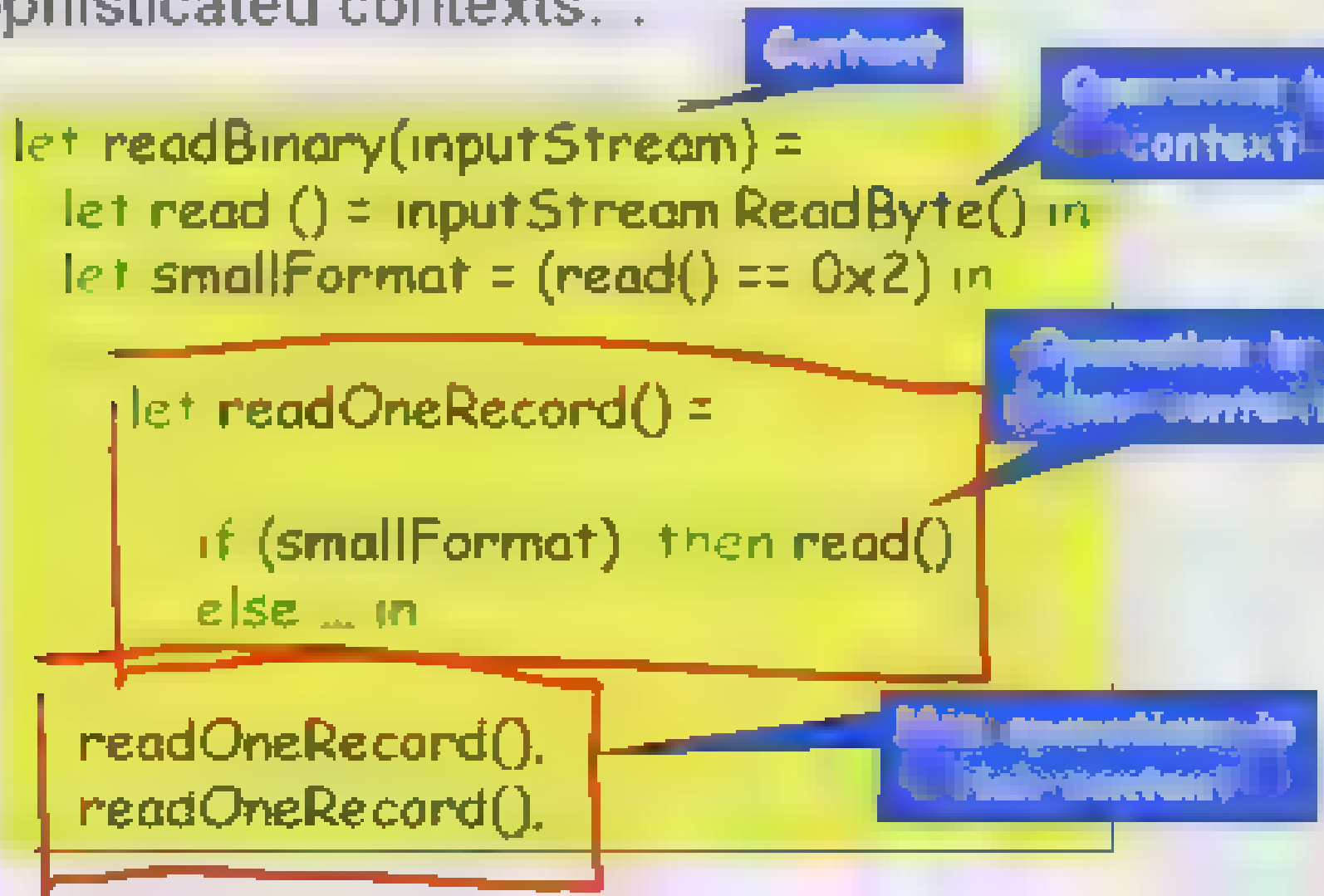
Orthogonal & Unified Constructs

- "let" + "capture". sophisticated operations in sophisticated contexts .



Orthogonal & Unified Constructs

- "let" + "capture": sophisticated operations in sophisticated contexts. .



Orthogonal & Unified Constructs

- Immutability the norm..

Orthogonal & Unified Constructs

- Immutability the norm..

```
let data = 3

do data := 4
```

Orthogonal & Unified Constructs

- Immutability the norm.

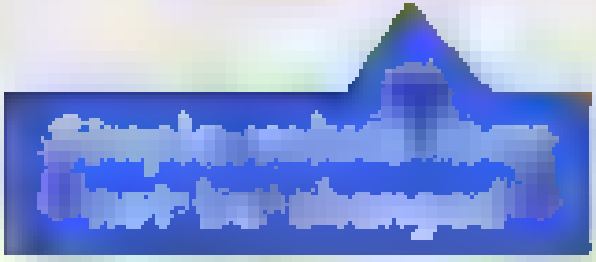
```
let data = 3

do data := 4
```

```
type data =
  { Name: string;
    Size: int;
    Items: Map<string,string> }

let NewData(name,items) =
  { Name=name;
    Size=Map.size items;
    Items=items }
```

Orthogonal & Unified Constructs

- Immutability the norm...


```
let data = 3

do data <- 4
```



```
type data =
  { Name: string,
    Size: int;
    Items: Map<string,string> }

let NewData(name,items) =
  { Name=name;
    Size=Map.size items,
    Items=items }
```

In praise of immutability

- Immutable objects can be relied on
- Immutable objects can transfer between threads
- Immutable objects can be aliased safely
- Immutable objects can be optimized

Orthogonal & Unified Constructs

- F# is a functional language
- F# is an imperative language
- **F# is a mixed functional/imperative language**
- F# is not Haskell. Loops encouraged

Orthogonal & Unified Constructs

- Function values simplify and unify
  - .iteration

Orthogonal & Unified Constructs

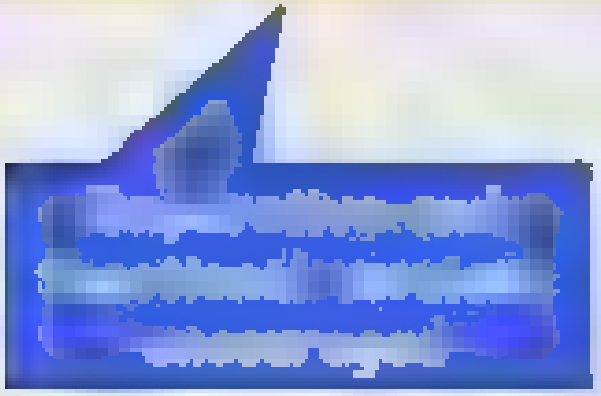
- Function values simplify and unify .iteration

```
val List.map: ('a -> 'b) -> list<'a> -> list<'b>
val List.foreach: list<'a> -> ('a -> unit) -> unit
val IEnumerable.map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>
val IEnumerable.foreach: IEnumerable<'a> -> ('a -> unit) -> unit
```

Orthogonal & Unified Constructs

- Function values simplify and unify .iteration

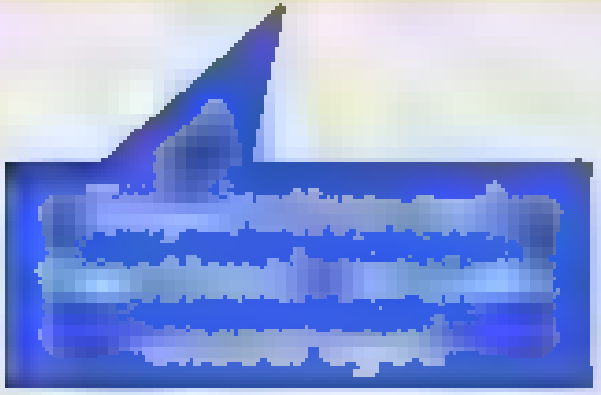
```
val List.map: ('a -> 'b) -> list<'a> -> list<'b>
val List.foreach: list<'a> -> ('a -> unit) -> unit
val IEnumerable.map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>
val IEnumerable.foreach: IEnumerable<'a> -> ('a -> unit) -> unit
```

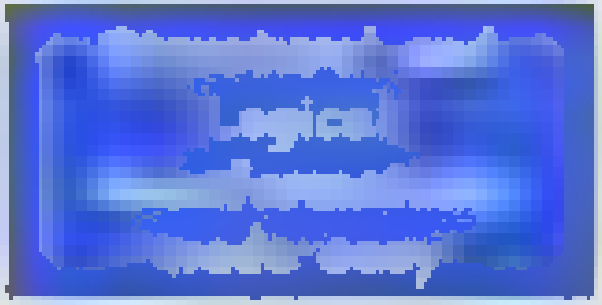


Orthogonal & Unified Constructs

- Function values simplify and unify
  .iteration

```
val List.map: ('a -> 'b) -> list<'a> -> list<'b>
val List.foreach: list<'a> -> ('a -> unit) -> unit
val IEnumerable.map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>
val IEnumerable.foreach: IEnumerable<'a> -> ('a -> unit) -> unit
```





Orthogonal & Unified Constructs

- Function values simplify and unify
  - .extension

```
type UnaryOperator =
  { evaluate: (float -> float),      // evaluation function
    differentiate: (expr -> expr), // symbolic differentiation function
    name: string }                 // identity

let sinop = { evaluate=sin,
              differentiate=Cos;
              name="sin" }
```

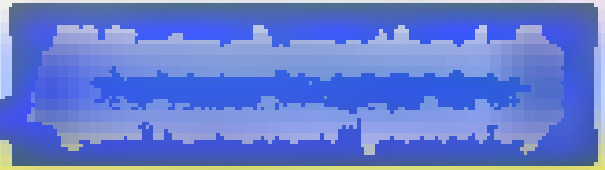
Orthogonal & Unified Constructs

- Function values simplify and unify
  - .extension

```
type UnaryOperator =
  { evaluate: (float -> float),      // evaluation function
    differentiate: (expr -> expr), // symbolic differentiation function
    name: string }                 // identity

let sinop = { evaluate=sin,
              differentiate=Cos;
              name="sin" }
```

Orthogonal & Unified Constructs

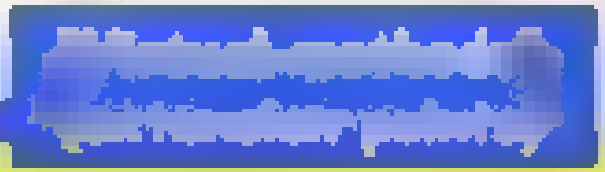
- Function values simplify and unify .extension



```
type UnaryOperator =
  { evaluate: (float -> float),      // evaluation function
    differentiate: (expr -> expr), // symbolic differentiation function
    name: string }                 // identity

let sinop = { evaluate=sin,
              differentiate=Cos;
              name="sin" }
```

Orthogonal & Unified Constructs

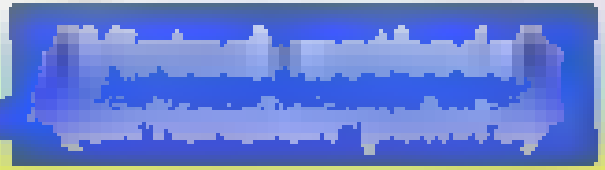
- Function values simplify and unify
  - .extension



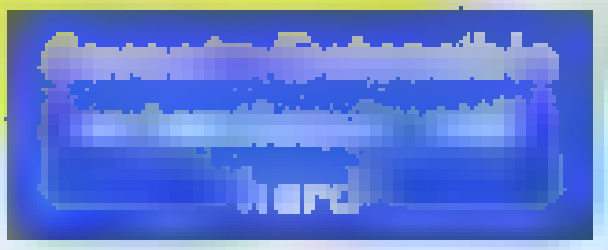
```
type UnaryOperator =
  { evaluate: (float -> float),       // evaluation function
    differentiate: (expr -> expr),    // symbolic differentiation function
    name: string }                    // identity

let sinop = { evaluate=sin,
              differentiate=Cos;
              name="sin" }
```

Orthogonal & Unified Constructs

- Function values simplify and unify
  - .extension



```
type UnaryOperator =
  { evaluate: (float -> float),       // evaluation function
    differentiate: (expr -> expr), // symbolic differentiation function
    name: string }                    // identity

let sinop = { evaluate=sin,
              differentiate=Cos;
              name="sin" }
```



Orthogonal & Unified Constructs

- Type parameters

```
Map<'a,'b>
List<'a>
Set<'a>
```

- Discriminated unions

```
type expr =
    | Sum of expr * expr
    | Prod of expr * expr
```

- Pattern matching
- Type inference
- Recursion (Mutually-referential objects)

Orthogonal & Unified Constructs

- Type parameters

```
Map<'a,'b>
List<'a>
Set<'a>
```

- Discriminated unions

```
type expr =
  | Sum of expr * expr
  | Prod of expr * expr
```

- Pattern matching

```
match expr with
  | Sum(a,b) ->
  | Prod(a,b) ->
```

- Type inference
- Recursion (Mutually-referential objects)

```
let rec map =
```

Less is More?

- Fewer concepts = Less time in class design
- No null pointers[1]
- Far fewer classes and other type definitions
- No constructors-calling-virtual-methods and other OO dangers

1 Except leaking across from .NET APIs

Problems are problems

- Unmanaged resources (IDisposable)
- Exceptions
- I/O, avoiding blocking, concurrency
- API design
- Complex software is, well, complex…

F# Observations

- F# gives you the tools to concentrate on the algorithmic complexities of symbolic processing

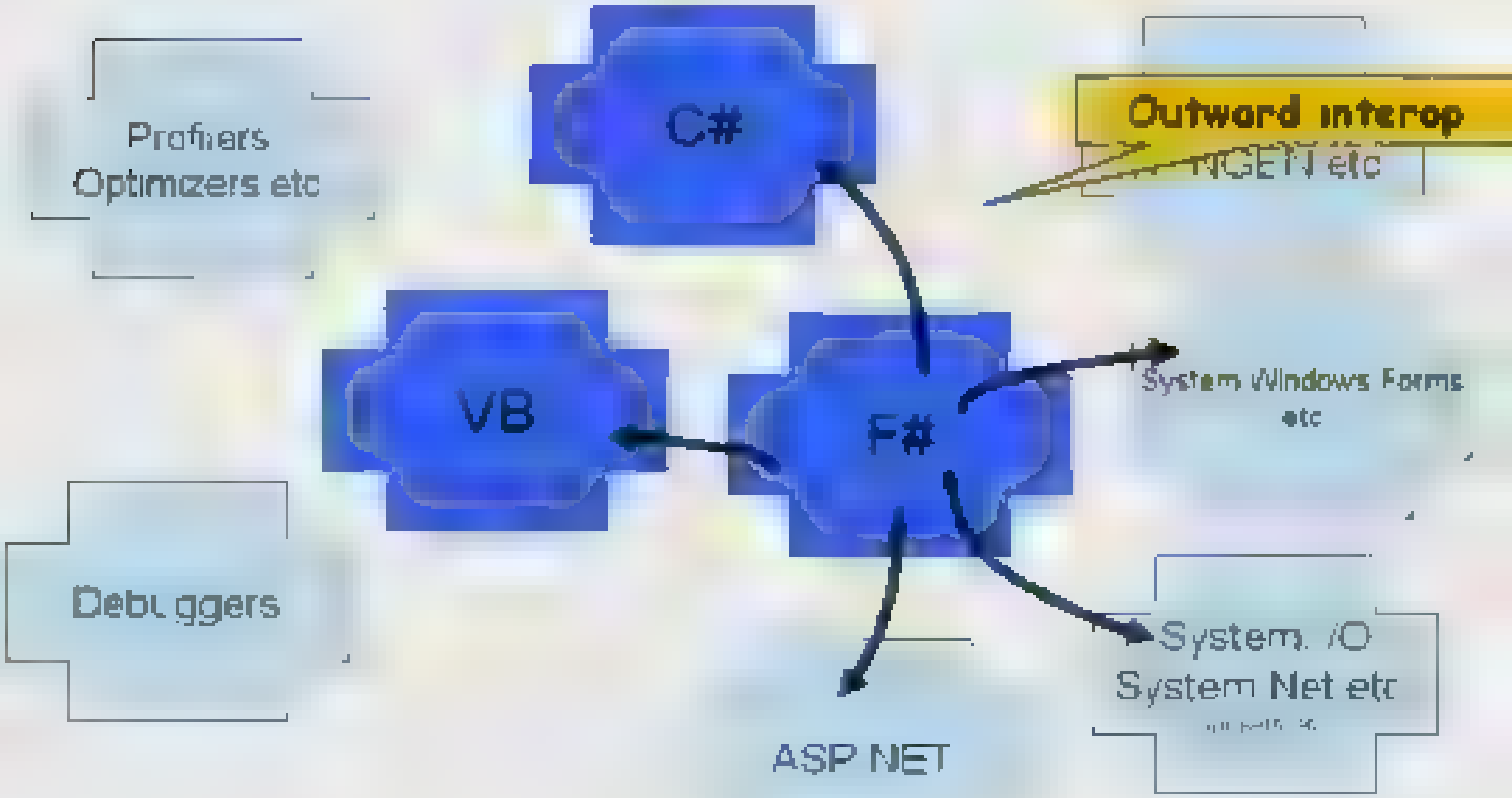
- F# is:
  - excellent for using .NET libraries
  - excellent for writing ML libraries
  - [illegible] ML libraries [illegible] from .NET

- So the niche seems to be for writing sophisticated applications
  - probably with some use of .NET components
  - probably with some symbolic processing
  - probably with some components written in F#

Part 2 F# Co-operates

The F# challenge: make it fit with .NET in a deeply practical way without losing the essential goodness of ML programming

Connecting F# to .NET



Outward Interop

- A very nice "." notation

```
System.Console.WriteLine("abc")

form.SetStyle(ControlStyles.AllPaintingInWmPaint, true);

form.ClientSize <- new System.Drawing.Size(292, 266)
```

- Quite similar to C#

- Delegates created using function values

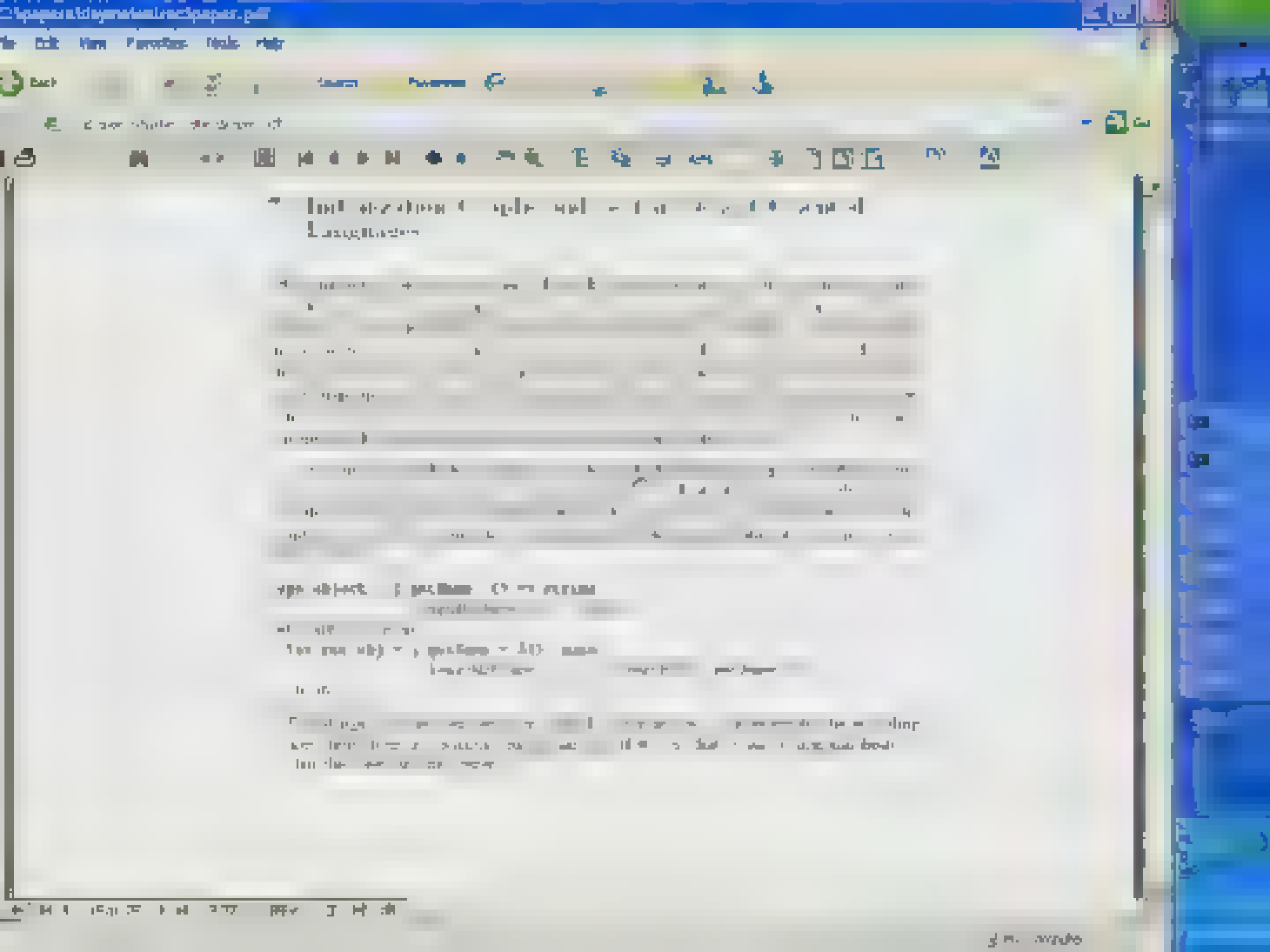
Outward Interop

- Object expressions == closures for objects

```
let myForm title n =
    ( new System.Windows.Forms.Form() as base
      with OnPaint(args) =
               base.OnPaint(args).
               Console.WriteLine("OnPaint\n")
      and OnResize(args) =
               base.OnResize(args).
               Console.WriteLine("OnResize  args = {0}\n" args)
    )
```

- Overriding only

- Orthogonal type inference, capture mutually recursive nested



Outward Interop

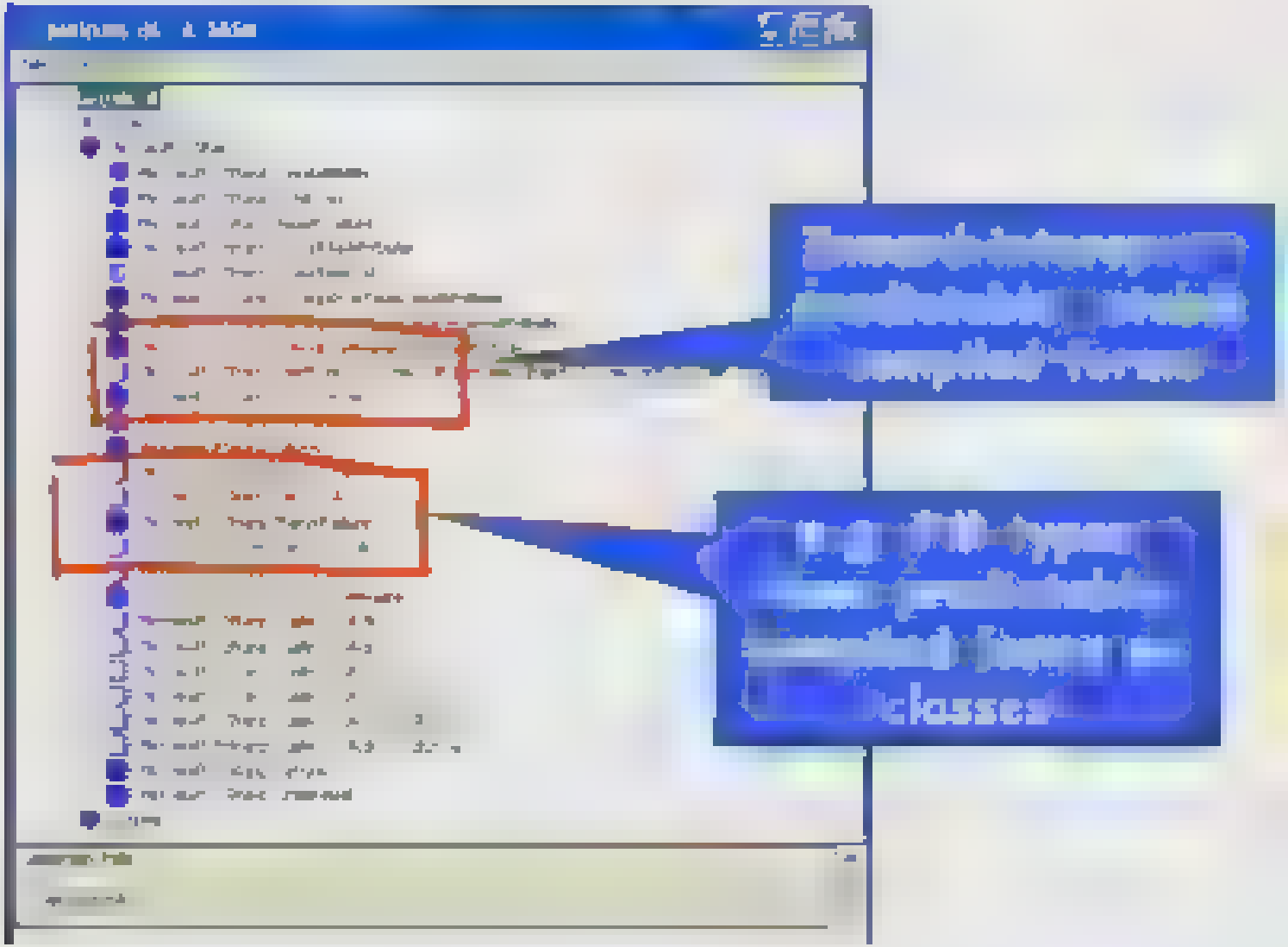
- Object expressions == closures for objects

```
let myForm title n =
    { new System.Windows.Forms.Form() as base
      with OnPaint(args) =
             base.OnPaint(args);
             Console.WriteLine("OnPaint\n")
      and OnResize(args) =
             base.OnResize(args);
             Console.WriteLine("OnResize: args = {0}\n", args)
    }
```

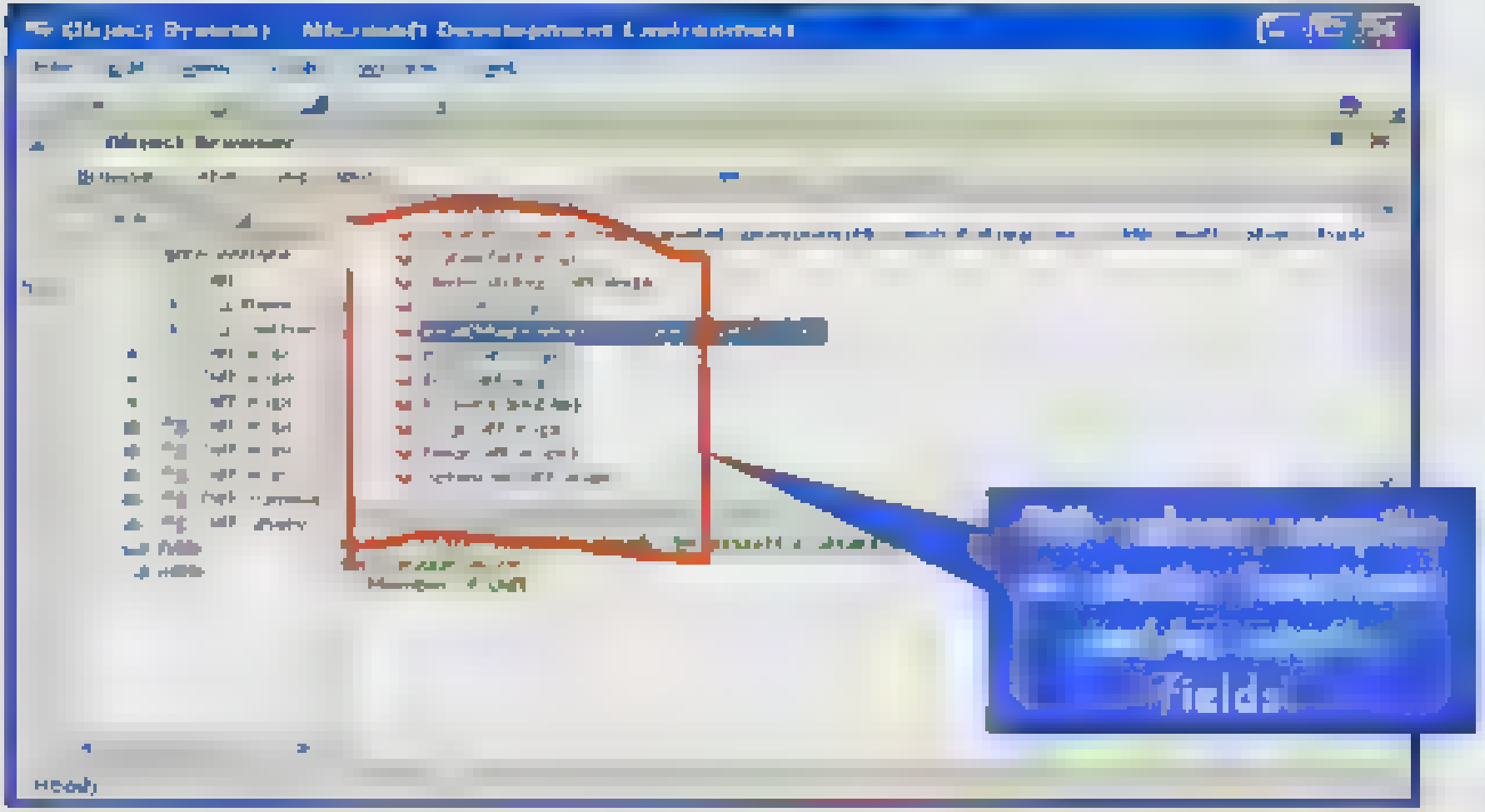
- Overriding only

- Orthogonal: type inference, capture, mutually recursive, nested

Inward Interop



Inward Interop



Inward Interop

- Inward Interop is **essential** for many reasons, e g. **componentization**, **testing**, **profiling**
- No other ML implementation in the world has it
- Inward interop gives you full access, but is sometimes a little awkward from C#
- Room for extensions to what is there

What else does F# offer? :)

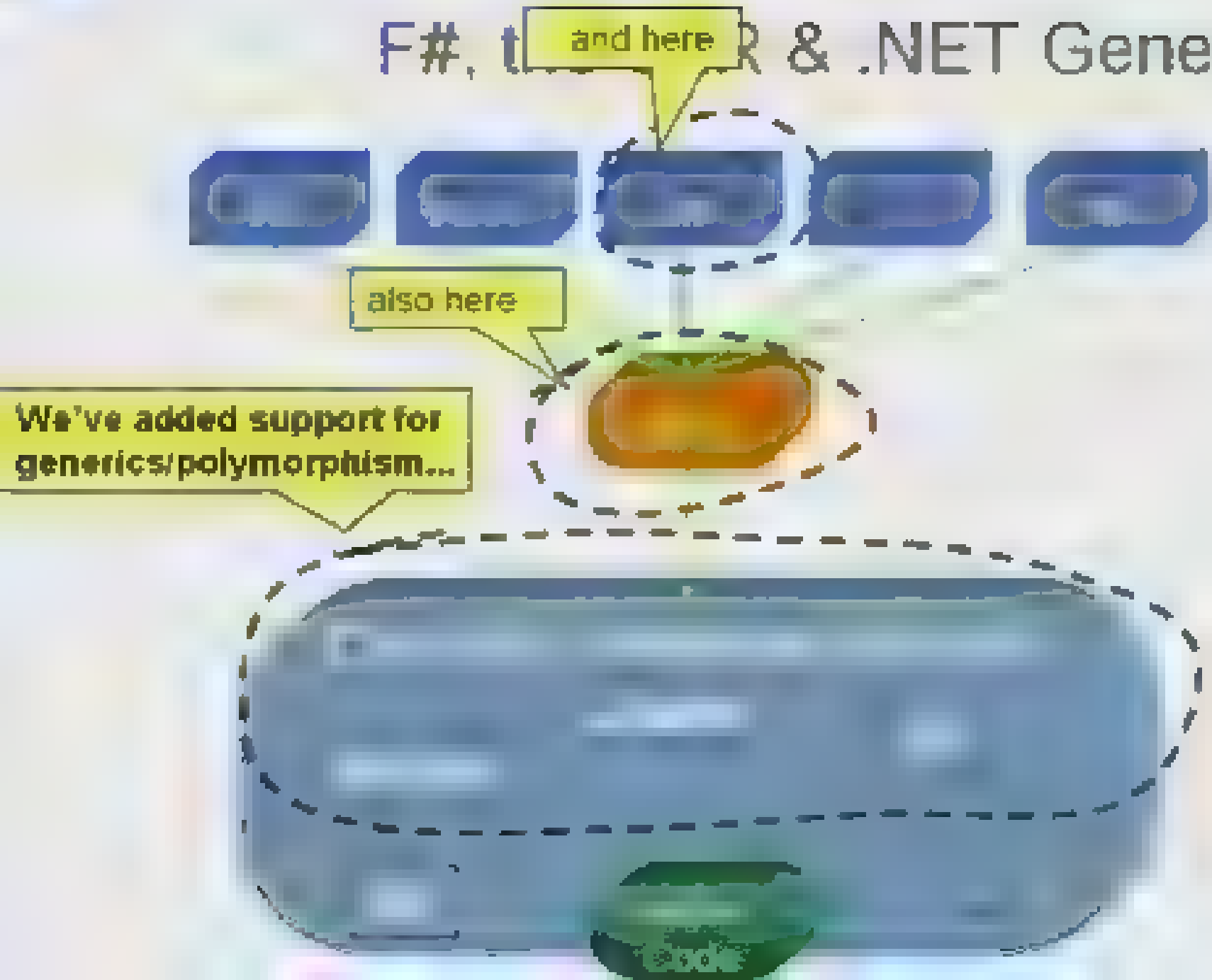
- Libraries galore
  - GUIs Network Speech Graphics
- Tools galore
  - CPU Profiling Memory Profiling Debugging
  - Visual Studio integration
- C# next door
  - Fantastic interop with C and COM
- Components
  - Can build versionable binary compatible DLLs
- Multi-threading that works
  - Even OCaml has problems here
- No significant runtime components
  - GC etc is not part of the package

What else does F# offer? :-)

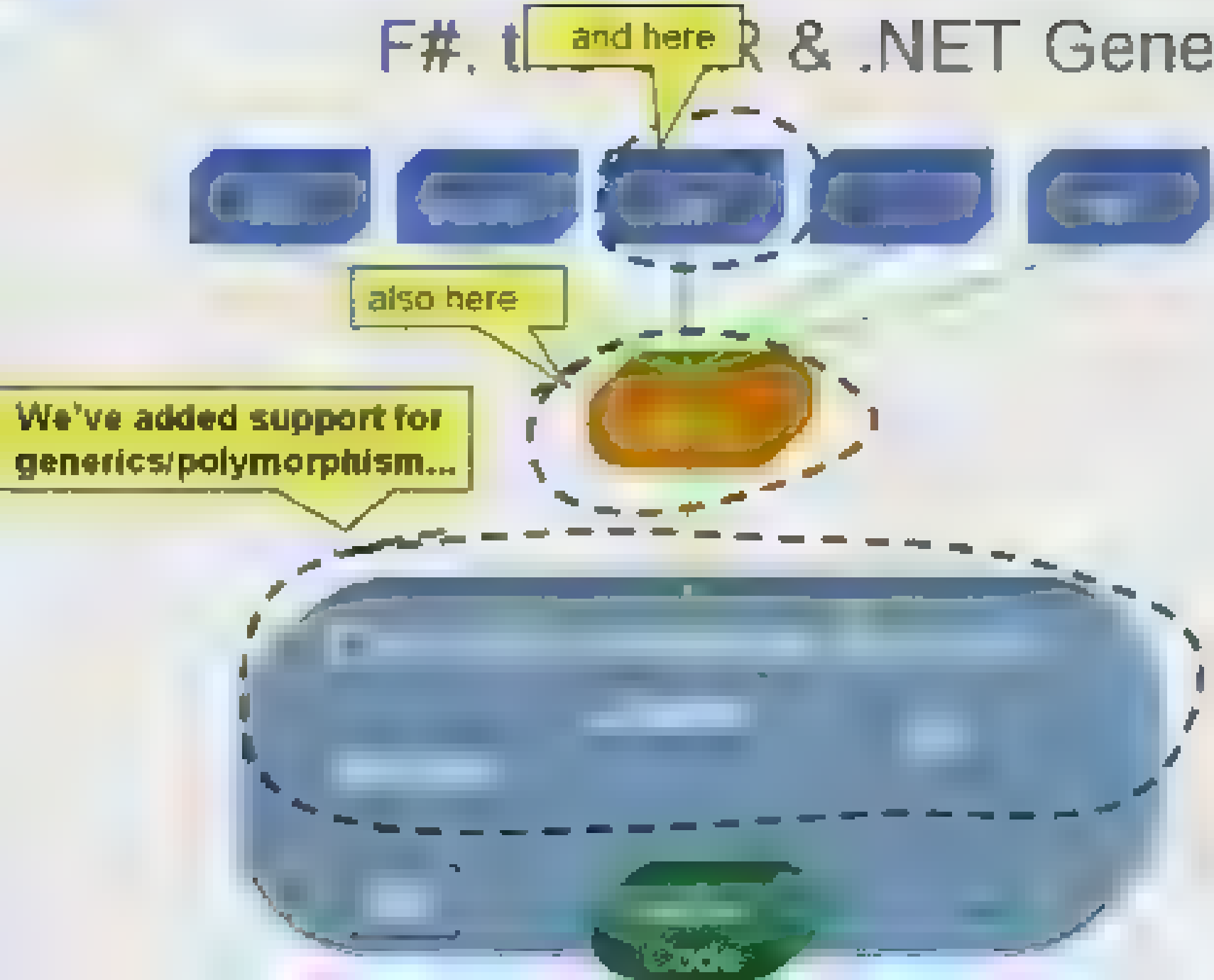
- Libraries galore
  - GUIs, Network, Speech, Graphics, .NET
- Tools galore
  - CPU Profiling, Memory Profiling, Debugging
  - Visual Studio integration
- C# next door
  - Fantastic interop with C and COM
- Components
  - Can build versionable, binary compatible DLLs
- Multi-threading that works
  - Even OCaml has problems here
- No significant runtime components
  - GC etc. is not part of the package

F#, the CLR & .NET Generics

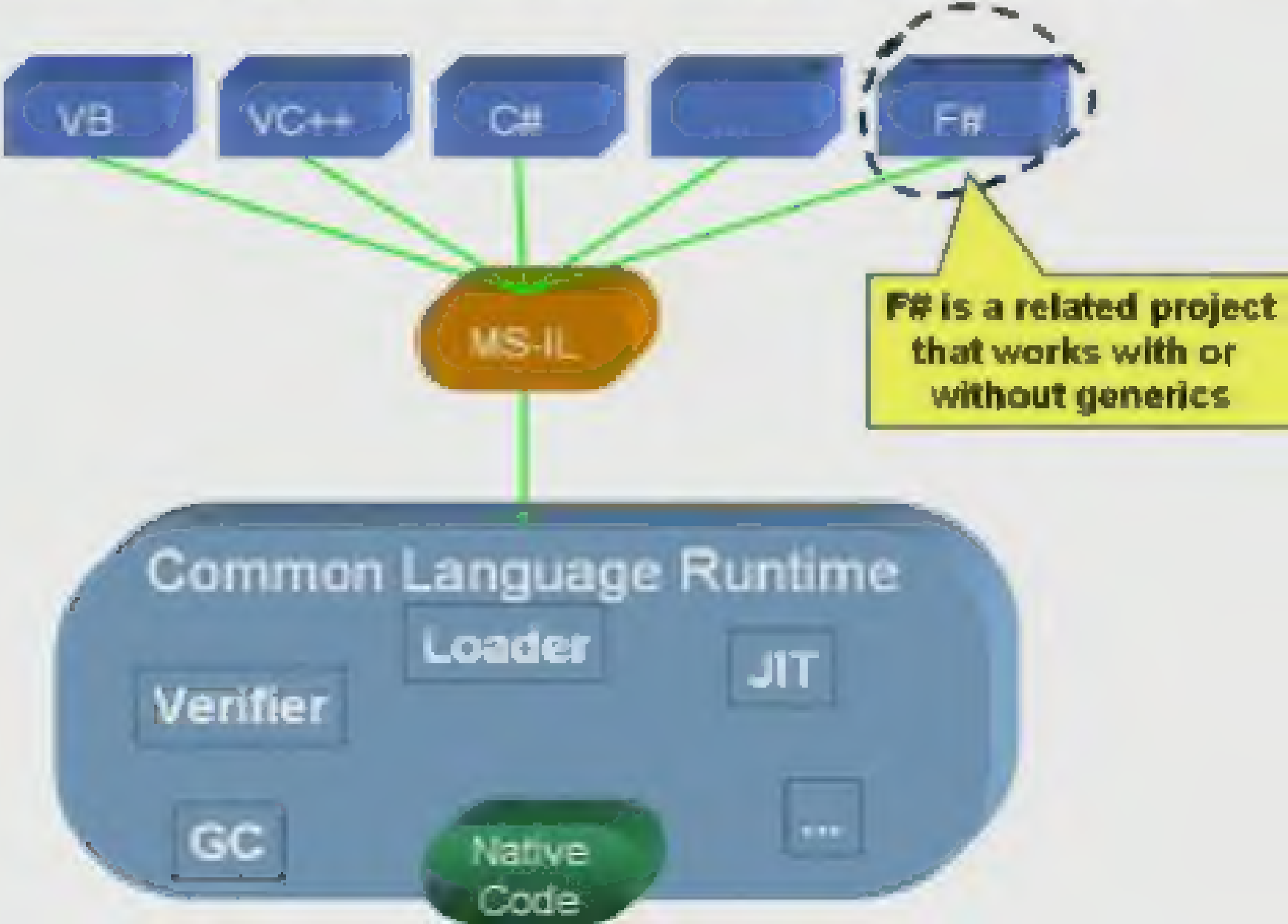
F#, t[illegible]R & .NET Generics



F#, t[illegible]R & .NET Generics



F#, the CLR & .NET Generics



F# v1.0

- Very stable core language and compiler
  - Some interesting language work will go on around the edges, e.g. ML-modules-without-ML-modules
- Being used by Zapato & SDV & many others
- ~6000 downloads/year
- VisualStudio 2005 Beta 1 integration
- ML compatibility library
- Samples etc.
- Tools: Lexer, Parser Generators

F# & Research

- Practical programming in ML still raises many interesting language design questions
- e.g. mutually recursive initialization that compensate correctly when failure occurs half-way through
- F# is a great place to conduct this research
  - easy to code examples of real-world practical programming
  - a simple clean language to extend

Questions?

http://research.microsoft.com/projects/fsharp

http://research.microsoft.com/projects/fsharp

autogroup: mailto:fsharp

\\cam-01-srv\dfsroot\projects\fcom\releases